Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
Кафедра «Информатика»

УТВЕРЖДАЮ
Заведующий кафедрой
_____ А. С. Кузнецов
«___» _____ 2019 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Инструментальное средство для выявления пропущенных вызовов библиотеки OpenGL с использованием методов машинного обучения

09.04.04 Программная инженерия

09.04.04.01 Программное обеспечение вычислительной техники и автоматизированных систем

Научный руководитель	_____	доцент, к.т.н.	А. С. Кузнецов
Выпускник	_____		А. М. Скрипачев
Рецензент	_____	доцент, к.т.н.	Е. П. Моргунов

Красноярск 2019

РЕФЕРАТ

Выпускная квалификационная работа по теме «Инструментальное средство для выявления пропущенных вызовов библиотеки OpenGL с использованием методов машинного обучения» содержит 70 страниц текстового документа, 24 использованных источника, 17 иллюстраций, 3 таблицы и 1 формулу.

СТАТИЧЕСКИЙ АНАЛИЗ КОДА, МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ, ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ, РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ, OPENGL.

Объект исследования – методы машинного обучения.

Предмет исследования – использование рекуррентных нейронных сетей для решения задачи поиска пропущенных библиотечных вызовов OpenGL.

Целью работы является разработка инструментального средства для выявления пропущенных вызовов библиотечных функций OpenGL на основе методов машинного обучения.

Задачи:

- проанализировать проблему поиска пропущенных вызовов библиотечных функций;

- разработать архитектуру нейронной сети на основе управляемого рекуррентного блока;

- сформировать обучающую и тестовую выборки из примеров исходного кода, в которых используются функции библиотеки OpenGL и проверить корректность обучения нейронной сети;

- разработать программное обеспечение для поиска пропущенных вызов библиотечных функций на основе разработанной нейронной сети;

- провести исследование работоспособности полученного программного обеспечения на главных сценариях использования.

В результате была построена модель искусственной нейронной сети, а также разработано и апробировано инструментальное средство для выявления пропущенных вызовов библиотеки OpenGL на основе построенной модели.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Программное обеспечение (ПО) вычислительных систем со временем становится труднее разрабатывать. Для упрощения разработки ПО создаются библиотеки функций, скрывающие сложность за программным интерфейсом. При использовании сторонних библиотек программисты зачастую применяют некоторые готовые решения — шаблоны, полученные из ресурсов интернета и литературы. При этом начинающие программисты имеют тенденцию пропускать различные части данных шаблонов при реализации собственных приложений. Помимо этого, проблемы, связанные с пропущенными вызовами, встречаются в обсуждениях на форумах, в отчетах об ошибках, в коммитах и в исходном коде. Из этого следует, что пропущенные вызовы функций могут быть источником дефектов программного обеспечения, которые нелегко обнаружить без вспомогательных инструментов.

В данной работе представлена гипотеза о том, что большинство подобных готовых решений содержат повторяющиеся шаблоны. Более того, данные шаблоны могут быть использованы для построения моделей, способных предсказать наличие (либо отсутствие) недостающих вызовов определенных библиотечных функций с использование машинного обучения.

Научная новизна заключается в применении нейронной сети, основанной на управляемом рекуррентном блоке, для поиска пропущенных вызовов библиотечных функций, что существенно отличается от имеющихся методов и средств решения задач данного класса.

Целью работы является разработка инструментального средства для выявления пропущенных вызовов библиотечных функций OpenGL на основе методов машинного обучения. Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать проблему поиска пропущенных вызовов библиотечных функций;

- разработать архитектуру нейронной сети на основе управляемого рекуррентного блока;
- сформировать обучающую и тестовую выборки из примеров исходного кода, в которых используются функции библиотеки OpenGL и проверить корректность обучения нейронной сети;
- разработать программное обеспечение для поиска пропущенных вызов библиотечных функций на основе разработанной нейронной сети;
- провести исследование работоспособности полученного программного обеспечения на главных сценариях использования.

1 Анализ методов и средств обеспечения надежности программ

1.1 Надежность программного обеспечения

Программное обеспечение (ПО) вычислительных систем становится все более значительным, сложным и опасным и его все труднее разрабатывать, но в то же время ПО все время упрощается, уменьшается в размерах, все легче поддается управлению.

С одной стороны, возрастают требования к программному обеспечению, связанные с усовершенствованием и усложнением операционных систем, аппаратных средств и интерфейса пользователя и с необходимостью внедрения современных информационных технологий, в первую очередь, сетевых. Внутреннее устройство программ, в связи с этим становится все более сложным, и возрастают требования к их надежности.

С другой стороны, накапливается и обобщается опыт разработки ПО, появляются все более гибкие и мощные методологии и средства, поддерживающие все этапы разработки ПО. Развивается методология визуального программирования и совершенствуются языки программирования. Совершенствование аппаратных средств ускоряет процессы компиляции, а также позволяет зачастую не беспокоиться об эффективности создаваемых программ.

Все происходящие в области развития вычислительной техники процессы оставляют неизменным одно: требование к высокой профессиональности программистов-разработчиков. Смещаются лишь акценты. Если на заре вычислительной техники программист-эксперт должен был во всех тонкостях разбираться в устройстве вычислительной установки, владеть системой команд процессора и изобретать «трюки», позволяющие экономить память и время, то современный специалист должен владеть методами и средствами грамотной организации своей работы, уметь работать в коллективе и должен программировать не столько «эффективно», сколько «надежно» [1].

Источниками ошибок в программном обеспечении являются специалисты – конкретные люди с их индивидуальными особенностями, квалификацией, талантом и опытом [2]. Вследствие этого плотность потоков ошибок и размеры необходимых корректировок в модулях и компонентах при разработке и сопровождении программного обеспечения могут различаться в десятки раз. Однако в крупных комплексах программ статистика и распределение ошибок и типов выполняемых изменений, необходимых для их исправления, для коллективов разных специалистов нивелируются и проявляются общие закономерности, которые могут использоваться как ориентиры при выявлении ошибок и их систематизации. Этому могут помогать оценки типовых ошибок, модификаций и корректировок путем их накопления и обобщения по опыту создания определенных классов программного обеспечения.

Оценка качества программного обеспечения могут проводиться с двух позиций: с позиции положительной эффективности и непосредственной адекватности их характеристик назначению, целям создания и применения, а также с негативной позиции возможного при этом ущерба – риска от использования ПС или системы. Показатели качества преимущественно отражают положительный эффект от применения программного обеспечения, и основная задача разработчиков проекта состоит в обеспечении высоких значений качества. Риски характеризуют возможные негативные последствия проявившихся в ходе эксплуатации ошибок или ущерб для пользователя при применении и функционировании программного обеспечения.

Качество программного обеспечения оценивается следующими характеристиками:

- функциональные возможности (Functionality). Набор атрибутов, относящихся к сути набора функций и их конкретным свойствам. Функциями являются те, которые реализуют установленные или предполагаемые потребности;

- надежность (Reliability). Набор атрибутов, относящихся к способности программного обеспечения сохранять свой уровень качества функционирования при установленных условиях за установленный период времени;

- практичность (Usability). Набор атрибутов, относящихся к объему работ, требуемых для использования и индивидуальной оценки такого использования определенным и предполагаемым кругом пользователей;

- эффективность (Efficiencies). Набор атрибутов, относящихся к соотношению между уровнем качества функционирования программного обеспечения и объемом используемых ресурсов при установленных условиях;

- сопровождаемость (Maintainability). Набор атрибутов, относящихся к объему работ, требуемых для проведения конкретных изменений (модификаций);

- мобильность (Portability). Набор атрибутов, относящихся к способности программного обеспечения быть перенесенным из одного окружения в другое [3].

В общем случае под ошибкой подразумевается неправильность, погрешность или неумышленное искажение объекта или процесса, что может быть причиной ущерба – риска при функционировании или применении программы [4]. При этом предполагается, что известно правильное, эталонное состояние объекта или процесса, по отношению к которому может быть определено наличие отклонения. Исходным эталоном для любого программного обеспечения являются спецификации требований заказчика или потенциального пользователя, предъявляемых к программам и ожидаемый пользователем или заказчиком эффект от использования программного обеспечения. Важной особенностью при этом является отсутствие полностью определенной программы – эталона, которой должны соответствовать текст и результаты функционирования разрабатываемой программы. Поэтому определить качество программного обеспечения и наличие ошибок в нем путем сравнения разрабатываемой программы с эталонной программной невозможно.

Исходя из определения ошибки в программном обеспечении, приведенном выше, можно сделать вывод, что ошибки, проявляющиеся в ходе функционирования программного обеспечения, могут влиять на все показатели качества. В работе рассматриваются ошибки, проявления которых влияют на надежность функционирования программного обеспечения.

По определению, установленному в ГОСТ 27.002-2015 [5], надежность – свойство объекта выполнять заданные функции, сохраняя во времени значения установленных эксплуатационных показателей в заданных пределах, соответствующим заданным режимам и условиям использования, технического обслуживания, ремонта, хранения и транспортирования.

При этом надежность является комплексным свойством, которое в зависимости от назначения объекта и условий его применения может включать безотказность, долговечность, ремонтопригодность и сохраняемость или определенные сочетания этих свойств (рисунок 1). Поскольку программное обеспечение в процессе эксплуатации не изнашивается, его поломка и ремонт в общепринятом смысле не производится, то надежность программного обеспечения имеет смысл характеризовать только с точки зрения.

Рисунок 1 – Надежность по ГОСТ 27.002-89

1.2 Ошибки в ПО

1.2.1 Классификация ошибок ПО

В борьбе со сложностью ПО используются две концепции:

- иерархическая структура. Иерархия позволяет разбить систему по уровням понимания (абстракции, управления). Концепция уровней позволяет анализировать систему, скрывая несущественные для данного уровня детали реализации других уровней. Иерархия позволяет понимать, проектировать и описывать сложные системы;

- независимость. В соответствии с этой концепцией, для минимизации сложности, необходимо максимально усилить независимость элементов системы. Это означает такую декомпозицию системы, чтобы её высокочастотная динамика была заключена в отдельных компонентах, а межкомпонентные взаимодействия (связи) описывали только низкочастотную динамику системы.

Методы обнаружения ошибок, которые базируются на введении в ПО системы различных видов избыточности:

- временная избыточность. Использование части производительности ЭВМ для контроля исполнения и восстановления работоспособности ПО после сбоя;

- информационная избыточность. Дублирование части данных информационной системы для обеспечения надежности и контроля достоверности данных;

- программная избыточность включает в себя: взаимное недоверие – компоненты системы проектируются, исходя из предположения, что другие компоненты и исходные данные содержат ошибки, и должны пытаться их обнаружить; немедленное обнаружение и регистрацию ошибок; выполнение одинаковых функций разными модулями системы и сопоставление результатов обработки; контроль и восстановление данных с использованием других видов избыточности.

Задача обеспечения ПО устойчивости к ошибкам направлены на применение методов минимизации ущерба, вызванного появлением ошибок, и включают в себя:

- обработку сбоев аппаратуры;
- повторное выполнение операций;
- динамическое изменение конфигурации;
- сокращенное обслуживание в случае отказа отдельных функций системы;
- копирование и восстановление данных;
- изоляцию ошибок.

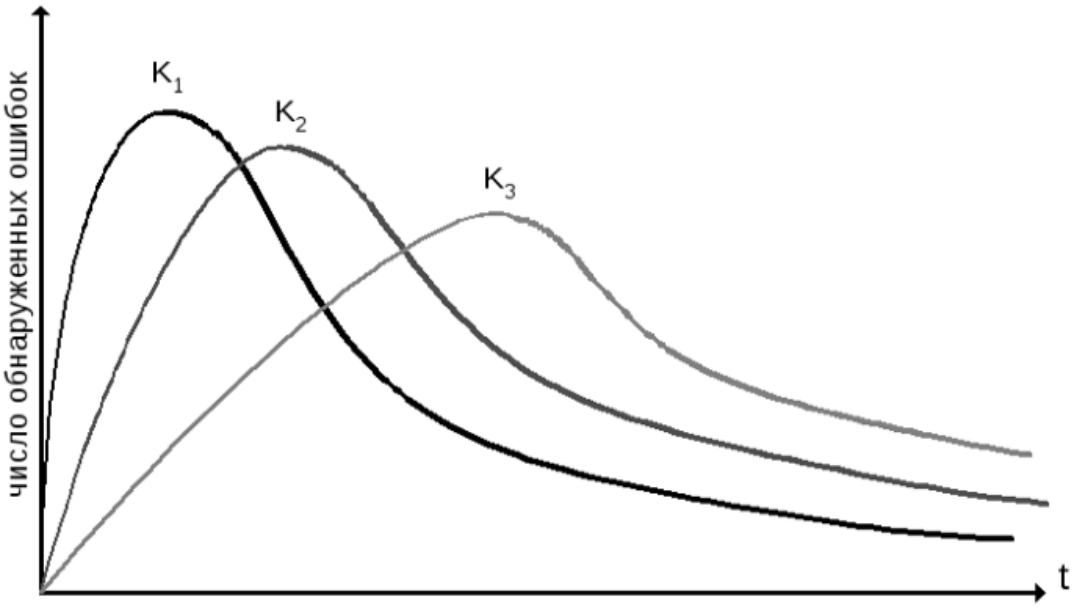
Дается 4 группы принципов обеспечения надежности:

- предупреждение ошибок;
- обнаружение ошибок;
- исправление ошибок;
- обеспечение устойчивости к ошибкам.

Действия, направленные на минимизацию ошибок и сбоев:

- предотвращение ошибок за счет структурного программирования;
- сокрытие информации или дозированный доступ к данным со стороны программных средств и объектов в объектно-ориентированном программировании;
- отладка;
- устойчивость к сбоям;
- обработка исключительных ситуаций (перехват ошибок, например, деление на ноль) и локализация ошибок и сбоев;
- восстановление программы после сбоя.

Чем интенсивнее использование программного изделия (ПИ), тем быстрее выявляются в нем ошибки. На рисунке приведена зависимость числа обнаруженных ошибок от числа использующих ПИ пользователей:



K – число пользователей, $K_1 > K_2 > K_3$.

Рисунок 2 – Интенсивность обнаружения ошибок от интенсивности использования

Процентные частоты появления ошибок в ПО по типам ошибок представлены в таблице 1.

Таблица 1 – Процентные частоты появления ошибок в ПО

Тип ошибки	Частота появления, %
Не полная или ошибочная спецификация	28
Отклонение от спецификации	12
Пренебрежение правилами программирования	10
Ошибочная выборка данных	11
Ошибочная логика или последовательность операций	12
Ошибочные арифметические операции	9
Нехватка времени для решения	5
Ошибка обработки прерываний	5
Неточная запись	8

1.2.2 Пропущенные вызовы функций как один из дефектов надежности программного обеспечения

Проблемы, связанные с пропущенными вызовами методов, всплывают на форумах, в отчетах об ошибках, в коммитах и в исходном коде. Из этого следует, что пропущенные вызовы функций могут быть источником дефектов программного обеспечения, которые нелегко обнаружить без вспомогательных инструментов.

Рассмотрим следующий пример: в Java фреймворке SWT у разработчика есть возможность создавать окна используя класс DialogPage напрямую, либо создав наследника этого класса. При генерации класс в среде разработки Eclipse получаем следующий код (листинг 1).

Листинг 1 – Код класса наследника DialogPage

```
1. public class MyPage extends DialogPage {
2.     @Override
3.     public void createControl(Composite parent) {
4.         // TODO Auto-generated method stub
5.     }
6. }
```

Допустим, нам необходимо создать единый контейнер (в SWT за это отвечает класс Composite) и, в дальнейшем, все содержимое диалогового окна помещать в этот контейнер. Для этого нам нужно написать следующий код в методе createControl (листинг 2).

Листинг 2 – Код переопределенного метода createControl

```
1. public void createControl(Composite parent) {
2.     Composite mycomp = new Composite(parent);
3. }
```

При запуске программы мы получим следующую ошибку: *an error has occurred. See error log for more details. org.eclipse.core.runtime.AssertionFailedException: null argument*.

При наследовании от класса, входящего в состав фреймворка, часто существуют зависимости вида «необходимо вызвать метод X при переопределении метода Y», которые необходимо учитывать при написании кода. Платформа пользовательского интерфейса Eclipse JFace ожидает, что класс, наследующий класс DialogPage вызовет метод setControl в переопределенном методе createControl. Однако в документации DialogPage не упоминается этот требование.

Описанный сценарий регулярно появляется на новостных форумах Eclipse newsgroup. Кроме того, полученная ошибка времени выполнения сама по себе довольно неоднозначна и может потребоваться время, чтобы понять и решить ее.

1.3 Методы обеспечения надежности программного обеспечения

К основным проблемам исследований надежности ПО относятся:

- разработка методов оценки и прогнозирования надежности ПО;
- определение основных факторов, влияющих на надежность ПО;
- разработка методов, обеспечивающих достижение заданного уровня надежности ПО;
- совершенствование методов повышения надежности ПО в процессе проектирования и эксплуатации.

В соответствии с [6] различают следующие виды работ, направленные на устранение ошибок в ПО:

- проверка;
- отладка;
- испытание программы.

1.3.1 Тестирование ПО

Важным этапом жизненного цикла ПО, определяющим качество и надежность системы, является тестирование. Тестирование – процесс

выполнения программ с намерением найти ошибки и включает в себя следующие этапы:

- автономное тестирование;
- тестирование сопряжений;
- тестирование функций;
- комплексное тестирование;
- тестирование полноты и корректности документации;
- тестирование конфигураций.

Надежность ПО повышается также с помощью применения различных методов тестирования. Полное тестирование ПО невозможно. Обычно применяют следующие виды тестирования:

- тестирование ветвей;
- математическое доказательство правильности алгоритма решения задачи (в некоторых работах именно в этом смысле употребляется слово верификация);
- символическое тестирование (или с помощью специально подобранных тестовых наборов), еще называется статическим тестированием. Удобно при локализации ошибки, проявление которой выявлено при конкретном узком или строго заданном диапазоне входных значений;
- динамическое тестирование (с помощью динамически генерируемых входных данных), что удобно при быстром тестировании во всем широком диапазоне входных параметров;
- тестирование путей выполнения программы;
- функциональное тестирование;
- проверки по времени выполнения программы;
- проверка по использованию ресурсов и стрессовое тестирование.

Одним из важных недостатков тестирования является его относительная дороговизна. Согласно статье «Подсчет себестоимости часа разработки программного обеспечения» [7] тестирование программного обеспечения занимает от 30 до 50 процентов от всей стоимости разработки. Для написания тестов необходимо тратить ресурсы разработчика, либо привлекать к работе

отдельного специалиста для проведения процесса тестирования. Данная проблема отчасти решается путем использования инструментальных средств для автоматической генерации тестов. К примеру, разработаны методы генерации, основанные на динамических символьных вычислениях [8], которые позволяют сократить время на написание тестов.

1.3.2 Статический анализ кода

Статический анализ кода – это процесс выявления ошибок и недочетов в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода.

Обзор кода (code review) – один из самых старых и надежных методов выявления дефектов. Он заключается в совместном внимательном чтении исходного кода и высказывании рекомендаций по его улучшению. В процессе чтения кода выявляются ошибки или участки кода, которые могут стать ошибочными в будущем. Также считается, что автор кода во время обзора не должен давать объяснений, как работает та или иная часть программы. Алгоритм работы должен быть понятен непосредственно из текста программы и комментариев. Если это условие не выполняется, то код должен быть доработан. Как правило, обзор кода хорошо работает, так как программисты намного легче замечают ошибки в чужом коде [9].

Единственный существенный недостаток методологии совместного обзора кода, это крайне высокая цена. Необходимо регулярно собирать нескольких программистов для обзора нового кода или повторного обзора кода после внесения рекомендаций. При этом программисты должны регулярно делать перерывы для отдыха. Если пытаться просматривать сразу большие фрагменты кода, то внимание быстро притупляется, и польза от обзора кода быстро сходит на нет.

Получается, что с одной стороны хочется регулярно осуществлять обзор кода, с другой – это слишком дорого. Компромиссным решением являются

инструменты статического анализа кода. Они обрабатывают исходные тексты программ и выдают программисту рекомендации обратить повышенное внимание на определенные участки кода. Конечно, программа не заменит полноценного обзора кода, выполняемого коллективом программистов. Однако соотношение польза/цена делает использование статического анализа весьма полезной практикой.

Задачи, решаемые программами статического анализа кода, можно разделить на 3 категории:

- выявление ошибок в программах;

- рекомендации по оформлению кода. Некоторые статические анализаторы позволяют проверять, соответствует ли исходный код, принятому в компании стандарту оформления кода. Имеется в виду контроль количества отступов в различных конструкциях, использование пробелов/символов табуляции и так далее;

- подсчет метрик. Метрика программного обеспечения – это мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций.

Как и у любой другой методологии выявления ошибок, у статического анализа есть свои сильные и слабые стороны. Важно понимать, что нет идеального метода тестирования программ. Для разных классов программного обеспечения разные методики будут давать разные результаты. Добиться высокого качества программы можно только используя сочетание различных методик.

Главное преимущество статического анализ состоит в возможности существенной снижении стоимости устранения дефектов в программе. Чем раньше ошибка выявлена, тем меньше стоимость ее исправления. Так согласно книге Стивена Макконнелла «Совершенный код. Практическое руководство по разработке программного обеспечения» [9], исправление ошибки на этапе тестирования обойдется в десять раз дороже, чем на этапе конструирования (написания кода). Эта зависимость отображена на рисунке 3:

Рисунок 3 – Средняя стоимость исправления дефектов в зависимости от времени их внесения и обнаружения

Инструменты статического анализа позволяют выявить большое количество ошибок этапа конструирования, что существенно снижает стоимость разработки всего проекта. Например, статический анализатор кода PVS-Studio может запускаться в фоновом режиме сразу после компиляции и в случае нахождения потенциальной ошибки уведомит программиста.

Другие преимущества статического анализа кода:

- полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получают управление крайне редко. Такие участки кода, как правило, не удается протестировать другими методами. Это позволяет находить дефекты в обработчиках редких ситуаций, в обработчиках ошибок или в системе логирования;

- статический анализ не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа. Это позволяет находить скрытые ошибки, которые могут проявить себя только через несколько лет. Например, это ошибки неопределенного поведения. Такие ошибки могут проявить себя при смене версии компилятора или при использовании других ключей для оптимизации кода;

- можно легко и быстро обнаруживать опечатки и последствия использования Copy-Paste. Как правило, нахождение этих ошибок другими способами является крайне неэффективной тратой времени и усилий. Обсуждая типовые ошибки, про такие ляпы, как правило, не вспоминают. Но на практике на их выявление тратится существенное время.

Недостатки статического анализа кода:

- статический анализ, как правило, слаб в диагностике утечек памяти и параллельных ошибок. Чтобы выявлять подобные ошибки, фактически необходимо виртуально выполнить часть программы. Это крайне сложно реализовать. Также подобные алгоритмы требуют очень много памяти и процессорного времени. Как правило, статические анализаторы ограничиваются диагностикой простых случаев. Более эффективным способом выявления утечек памяти и параллельных ошибок является использование инструментов динамического анализа;

- программа статического анализа предупреждает о подозрительных местах. Это значит, что на самом деле код, может быть совершенно корректен. Это называется ложно-позитивными срабатываниями. Понять, указывает анализатор на ошибку или выдал ложное срабатывание, может только программист. Необходимость просматривать ложные срабатывания отнимает рабочее время и ослабляет внимание к тем участкам кода, где в действительности содержатся ошибки.

1.4 Анализ существующих решений

1.4.1 Система DMMC

Система DMMC – это система обнаружения пропущенных вызовов методов [10]. Она представляет из себя статический анализатор кода, который определяет список мест в коде, где могут быть пропущенные вызовы методов.

Идея обнаружения пропущенного вызова состоит в том, что во фрагменте кода, вероятно, содержится дефект, если количество фрагментов кода, отличающихся от исходного вызываемыми методами, намного превосходит количество фрагментов кода идентичных имеющемуся с точки зрения последовательности методов. В статье «Detecting Missing Method Calls in Object-Oriented Software» [10] приводится следующая непрограммная иллюстрация этой идеи.

Предположим, в некотором ресторане, есть одно место X с одной вилкой и одной ложкой. Во всем ресторане есть одно другое место с одной вилкой и одной ложкой (цвет ложки, к примеру, может быть другим) и есть 99 других мест с одной вилкой, одной ложкой и одним ножом. С точки зрения алгоритма, на месте X должен присутствовать нож.

В рассматриваемой системе есть три базовых понятия:

- фрагмент кода с вызовом методов одного объекта (type-usage), в дальнейшем – фрагмент кода объекта;

- идентичный фрагмент используемого типа данных;

- схожий фрагмент используемого типа данных.

Фрагмент кода объекта – это набор вызовов методов объекта в рамках одного фрагмента исходного кода. Фрагменты являются идентичными, если наборы методов в них совпадают. Фрагменты являются схожими если наборы методов в них совпадают за исключением одного метода.

Алгоритм работы системы:

а) получение из исходного кода всех фрагментов кода объекта;

б) для каждого полученного фрагмента x;

1) определить количество фрагментов, идентичных *x*;

2) определить количество фрагментов, схожих *x*;

3) вычислить показатель S-score по формуле (1);

4) если S-score < 1 – сформировать список потенциально пропущенных методов для фрагмента x;

в) сформировать список фрагментов со списками пропущенных методов, отсортированных по убыванию по показателю S-score.

$$S - score(x) = 1 - \frac{|E(x)|}{|E(x)+A(x)|}, \quad (1)$$

где E(x) – количество фрагментов, идентичных x;

A(x) – количество фрагментов, схожих с x;

x – исходный фрагмент из полученного списка на шаге (а).

Плюсы системы:

- низкий процент ложноположительных срабатываний;
- высокая точность получаемых результатов.

Минусы системы:

- инструмент работает только с исходным кодом на языке Java;
- алгоритм работы системы рассчитан на работу только с объектно-ориентированными языками программирования;
- инструмент корректно работает если пропущено не больше одного метода.

1.4.2 Система PR-Miner

Система PR-Miner предназначена для выявления зависимостей (правил) между вызовами функций в программном коде [11]. В системе задача выявления зависимостей решается как задача поиска ассоциативных правил. Однако, систему можно рассматривать лишь как вспомогательный инструмент, так как она сама по себе не определяет факт наличия/отсутствия пропущенного вызова.

Поиск ассоциативных правил – это метод обучения машин на базе правил обнаружения интересующих нас связей между переменными в наборе данных [12]. Метод предлагается для установления сильных правил, обнаруженных в наборе данных с помощью некоторых мер интересности. Этот основанный на правилах подход генерирует также новые правила по мере анализа

дополнительных данных. Конечной целью, исходя из достаточно большого набора данных, является помощь машине имитировать выделение признаков и создание возможности нахождения абстрактных ассоциаций из новых неклассифицированных данных.

Для поиска ассоциативных правил в системе используется алгоритм FPClose [13], который является одним из самых эффективных методов для решения задачи поиска ассоциативных правил.

Общая схема работы система из статьи [11] изображена на рисунке 4.

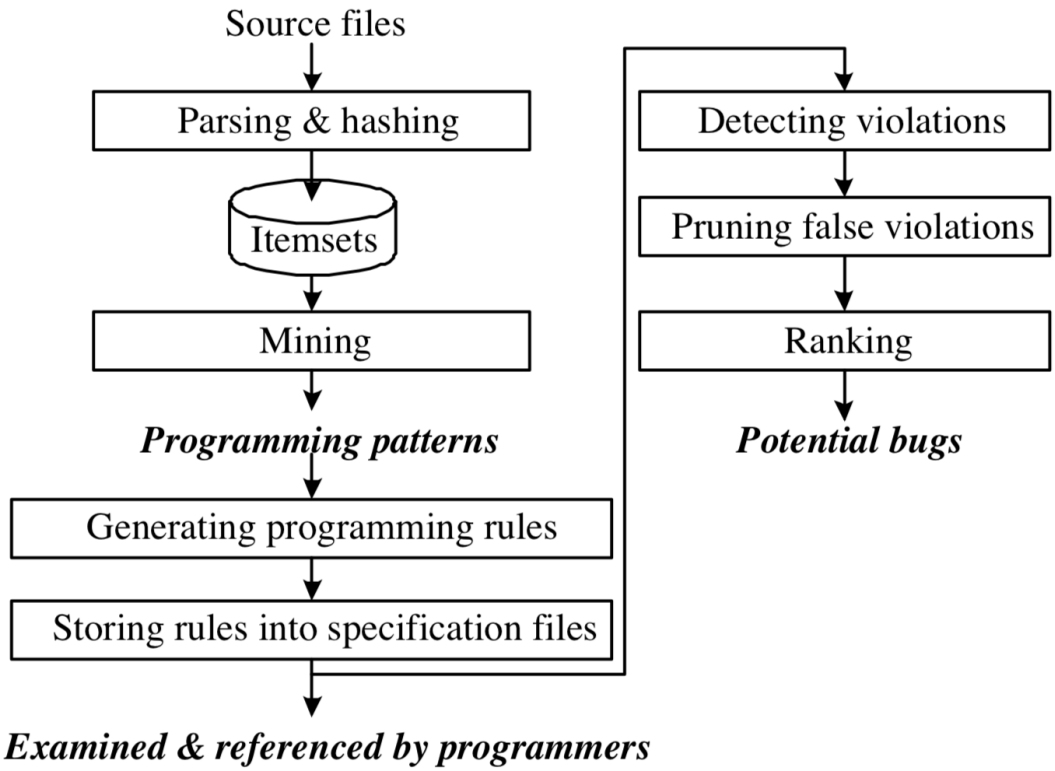


Рисунок 4 – Общая схема работы системы

Плюсы системы:

- высокая скорость работы;

- высокая точно определения зависимостей между функциями.

Минусы системы:

- возможно лишь использование как вспомогательного инструмента при поиске пропущенных вызовов;

- для корректной определения зависимостей необходим большой объем данных.

1.5 Предлагаемое решение

Исходя из недостатков рассмотренных систем и учитывая их положительные стороны, необходимо, чтобы разрабатываемая система соответствовала следующим показателям:

- низкий процент ложноположительных и ложноотрицательных срабатываний;
- высокая точность определения пропущенного вызова;
- потенциальная возможность для использования в разных языках программирования;
- определение сразу нескольких пропущенных вызовов (если такие имеются).

Постановка задачи: имеется фрагмент кода F, состоящий из вызовов функций и других языковых конструкций. Множество функций, которые могут входить в F конечно и заранее определено. Необходимо определить множество вызовов функций, которые были пропущены в фрагменте F. Если пропущенных вызовов не было, то искомое множество – это пустое множество.

Для одного пропущенного вызова задача схожа с задачей классификации. Ее общая постановка звучит следующим образом: имеется множество объектов (ситуаций), разделенных некоторым образом на классы. Задано конечное множество объектов, для которых известно, к каким классам они относятся. Это множество называется выборкой. Классовая принадлежность остальных объектов неизвестна. Требуется построить алгоритм, способный классифицировать произвольный объект из исходного множества [14]. В нашем случае в качестве множества объектов у нас выступает наборы функций из фрагментов кода, в качестве классов – набор всех возможных функций плюс отдельных класс, который будет отображать отсутствие пропущенной функции.

Задачу классификации будем решать средствами машинного обучения. Машинное обучение – класс методов искусственного интеллекта, характерной чертой которых является не прямое решение задачи, а обучение в процессе применения решений множества сходных задач. Для построения таких методов используются средства математической статистики, численных методов, методов оптимизации, теории вероятностей, теории графов, различные техники работы с данными в цифровой форме [15].

Очевидно, что задача классификации подразумевает заранее определенное множество допустимых функций. Значит для исследования необходимо выбрать фреймворк/инструмент, из которого можно будет получить все используемые в нем функции для формирования множества допустимых функций.

В качестве объекта исследования выбрана библиотека OpenGL. В дальнейшем, в работе все функции, входящие в состав библиотеки, будет называть библиотечными функциями, а их вызовы – вызовами библиотечных функций. Эта библиотека очень удобна для исследования, потому что:

- имеется ограниченное количество библиотечных функций, что позволит нам определить полный список классов;

- вызовы входящих в нее функций образуют шаблоны;

- для данной библиотеки доступен широкий набор примеров в открытых источниках.

2 Разработка алгоритма статистического анализа для поиска пропущенных вызовов функций на основе методов машинного обучения

2.1 Общая схема работы системы

На рисунке 5 изображена общая схема работы системы.

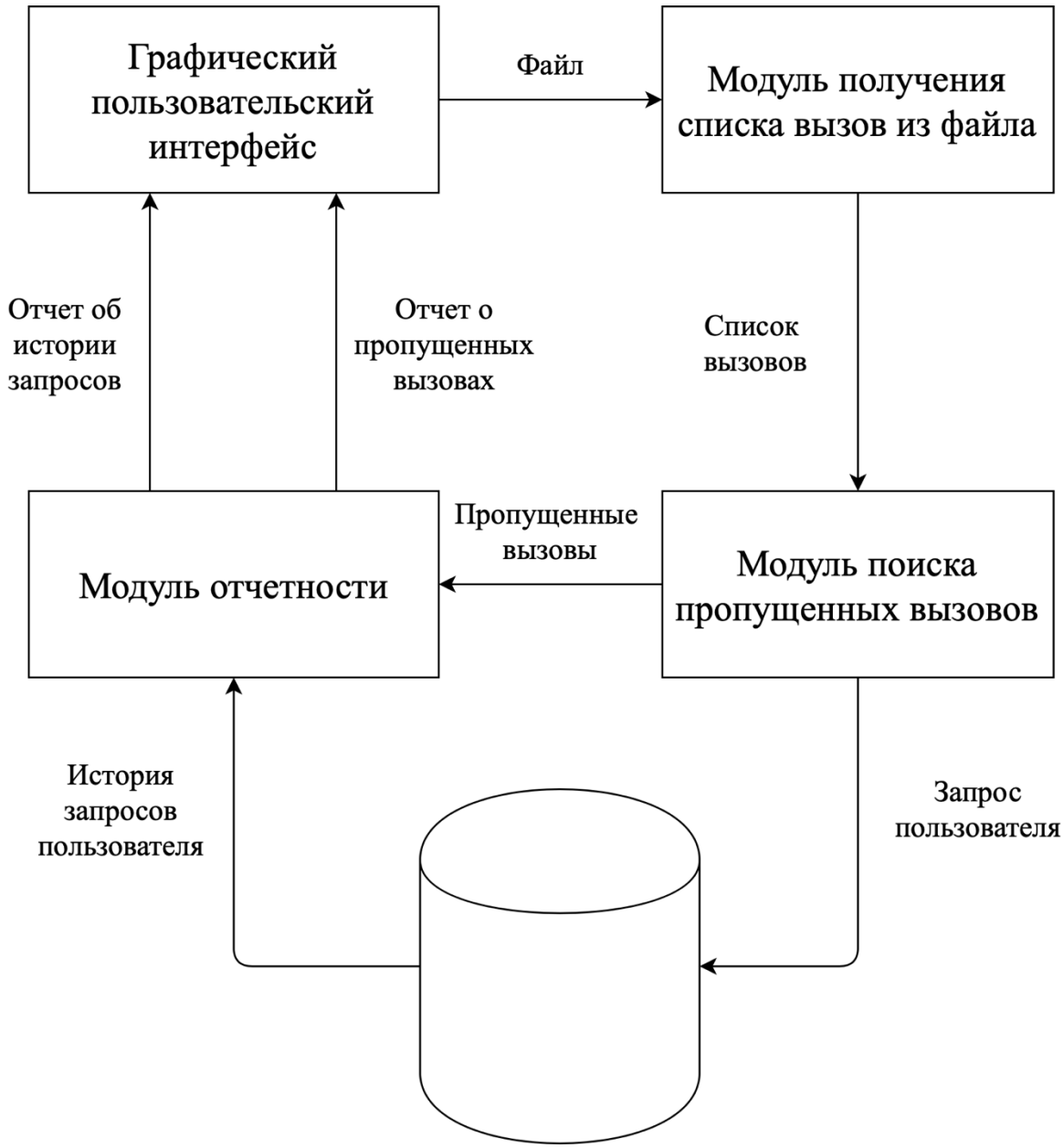


Рисунок 5 – Общая схема работы системы

Задача модуля извлечения библиотечных вызовов функций: получить список функций в исходном коде с содержащимися в них вызовами библиотечных функций.

Схема базы данных представлена на рисунке 6.

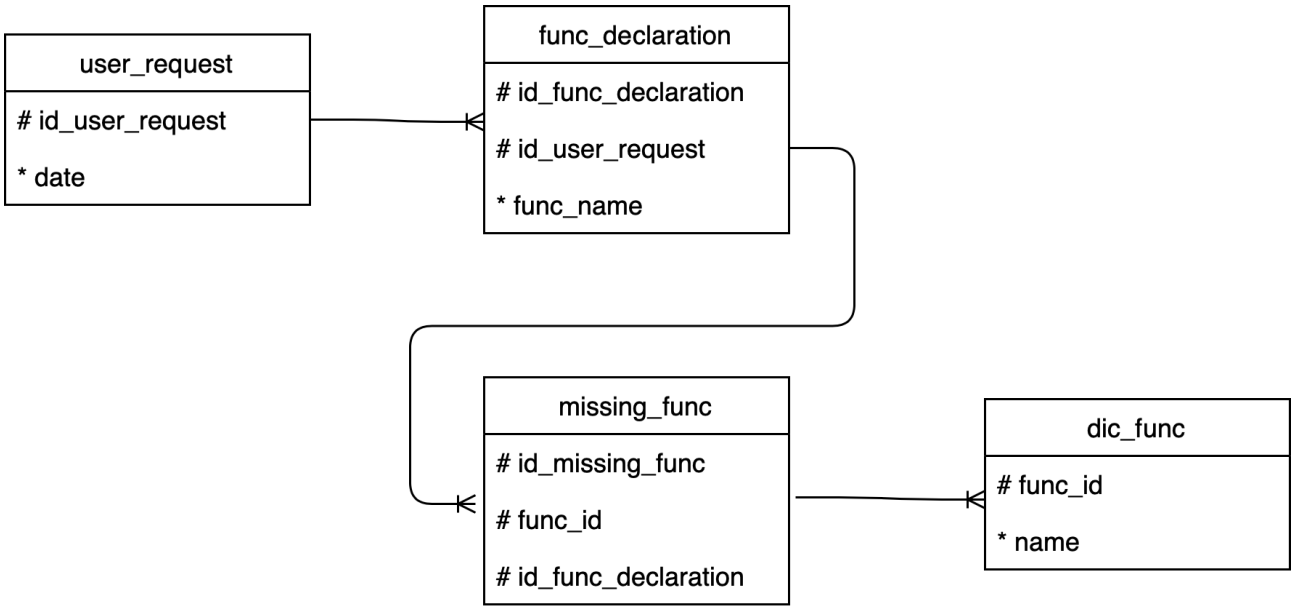


Рисунок 6 – Схема базы данных

Общая схема работы модуля в виде блок схемы, составленной в соответствии с ГОСТ 19.701-90 [16] изображена на рисунке 7.

Рисунок 7 – Алгоритм работы модуля извлечения вызов

Для того чтобы облегчить выделение списка функций и вызовов библиотечных функций, можно транслировать исходный код в промежуточное представление. После преобразования кода в промежуточное представление необходимо удалить из него все языковые конструкции кроме объявлений и вызовов функций. Затем необходимо получить список всех объявленных функций и каждую проверить – есть ли в ней вызовы библиотечных функций. Если есть, помещаем ее в результирующий список, при этом прикрепляя к этой функции все вызовы библиотечных функций, которые в ней содержались.

Задача модуля анализа пропущенных вызовов: для каждого полученного объявления функций определить – пропущены ли в нем вызовы библиотечных функций или нет, и, если пропущены – выдать их список. Как уже было сказано ранее, для нескольких пропущенных вызовов система должна работать итеративным путем. Общая схема работы алгоритма изображена на рисунке 8.

Рисунок 8 – Алгоритм работы модуля поиска пропущенных вызовов

Задача модуля обработки отчетности: форматирование информации о пропущенных вызовах в удобном для пользователя виде.

2.2 Нейронные сети

2.2.1 Общая информация о нейронных сетях

Искусственная нейронная сеть (ИНС) – математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей – сетей нервных клеток живого организма. Это понятие возникло при изучении процессов, протекающих в мозге, и при попытке смоделировать эти процессы. Первой такой попыткой были нейронные сети У. Маккалока и У. Питтса. После разработки алгоритмов обучения получаемые модели стали использовать в практических целях: в задачах прогнозирования, для распознавания образов, в задачах управления и др. [17].

ИНС представляет собой систему соединённых и взаимодействующих между собой простых процессоров (искусственных нейронов). Такие процессоры обычно довольно просты (особенно в сравнении с процессорами, используемыми в персональных компьютерах). Каждый процессор подобной сети имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он периодически посылает другим процессорам. И, тем не менее, будучи соединёнными в достаточно большую сеть с управляемым взаимодействием, такие по отдельности простые процессоры вместе способны выполнять довольно сложные задачи.

- с точки зрения машинного обучения, нейронная сеть представляет собой частный случай методов распознавания образов, дискриминантного анализа, методов кластеризации и т. п.;

- с математической точки зрения, обучение нейронных сетей – это многопараметрическая задача нелинейной оптимизации;

- с точки зрения кибернетики, нейронная сеть используется в задачах адаптивного управления и как алгоритмы для робототехники;

- с точки зрения развития вычислительной техники и программирования, нейронная сеть – способ решения проблемы эффективного параллелизма;

- с точки зрения искусственного интеллекта, ИНС является основой философского течения коннективизма и основным направлением в структурном подходе по изучению возможности построения (моделирования) естественного интеллекта с помощью компьютерных алгоритмов.

Нейронные сети не программируются в привычном смысле этого слова, они обучаются. Возможность обучения – одно из главных преимуществ нейронных сетей перед традиционными алгоритмами. Технически обучение заключается в нахождении коэффициентов связей между нейронами. В процессе обучения нейронная сеть способна выявлять сложные зависимости между входными данными и выходными, а также выполнять обобщение. Это значит, что в случае успешного обучения сеть сможет вернуть верный результат на основании данных, которые отсутствовали в обучающей выборке, а также неполных и/или «зашумленных», частично искаженных данных.

Задачи, решаемые нейронными сетями:

- распознавание образов и классификация;
- принятие решений и управление;
- кластеризация;
- прогнозирование;
- аппроксимация;
- сжатие данных и ассоциативная память;
- анализ данных;
- оптимизация.

Рассмотрим этапы решения задач с использованием нейронных сетей.

2.2.1.1 Сбор данных для обучения

Выбор данных для обучения сети и их обработка является самым сложным этапом решения задачи. Набор данных для обучения должен удовлетворять нескольким критериям:

- репрезентативность – данные должны иллюстрировать истинное положение вещей в предметной области;

- непротиворечивость – противоречивые данные в обучающей выборке приведут к плохому качеству обучения сети.

Исходные данные преобразуются к виду, в котором их можно подать на входы сети. Каждая запись в файле данных называется обучающей парой или обучающим вектором. Обучающий вектор содержит по одному значению на каждый вход сети и, в зависимости от типа обучения (с учителем или без), по одному значению для каждого выхода сети. Обучение сети на «сыром» наборе, как правило, не даёт качественных результатов. Существует ряд способов улучшить «восприятие» сети:

- нормировка выполняется, когда на различные входы подаются данные разной размерности. Например, на первый вход сети подаются величины со значениями от нуля до единицы, а на второй – от ста до тысячи. При отсутствии нормировки значения на втором входе будут всегда оказывать существенно большее влияние на выход сети, чем значения на первом входе. При нормировке размерности всех входных и выходных данных сводятся воедино;

- квантование выполняется над непрерывными величинами, для которых выделяется конечный набор дискретных значений. Например, квантование используют для задания частот звуковых сигналов при распознавании речи;

- фильтрация выполняется для «зашумленных» данных.

Кроме того, большую роль играет само представление как входных, так и выходных данных. Предположим, сеть обучается распознаванию букв на изображениях и имеет один числовой выход – номер буквы в алфавите. В этом случае сеть получит ложное представление о том, что буквы с номерами 1 и 2

более похожи, чем буквы с номерами 1 и 3, что, в общем, неверно. Для того, чтобы избежать такой ситуации, используют топологию сети с большим числом выходов, когда каждый выход имеет свой смысл. Чем больше выходов в сети, тем большее расстояние между классами и тем сложнее их спутать.

2.2.1.2 Выбор топологии сети

Выбирать тип сети следует, исходя из постановки задачи и имеющихся данных для обучения. Для обучения с учителем требуется наличие для каждого элемента выборки «экспертной» оценки. Иногда получение такой оценки для большого массива данных просто невозможно. В этих случаях естественным выбором является сеть, обучающаяся без учителя (например, самоорганизующаяся карта Кохонена или нейронная сеть Хопфилда). При решении других задач (таких, как прогнозирование временных рядов) экспертная оценка уже содержится в исходных данных и может быть выделена при их обработке.

2.2.1.3 Экспериментальный подбор характеристик сети

После выбора общей структуры нужно экспериментально подобрать параметры сети. Для сетей, подобных перцептрону, это будет число слоев, число блоков в скрытых слоях (для сетей Ворда), наличие или отсутствие обходных соединений, передаточные функции нейронов. При выборе количества слоев и нейронов в них следует исходить из того, что способности сети к обобщению тем выше, чем больше суммарное число связей между нейронами. С другой стороны, число связей ограничено сверху количеством записей в обучающих данных.

2.2.1.4 Экспериментальный подбор параметров обучения

После выбора конкретной топологии необходимо выбрать параметры обучения нейронной сети. Этот этап особенно важен для сетей, обучающихся с учителем. От правильного выбора параметров зависит не только то, насколько быстро ответы сети будут сходиться к правильным ответам. Например, выбор низкой скорости обучения увеличит время схождения, однако иногда позволяет избежать паралича сети. Увеличение момента обучения может привести как к увеличению, так и к уменьшению времени сходимости, в зависимости от формы поверхности ошибки. Исходя из такого противоречивого влияния параметров, можно сделать вывод, что их значения нужно выбирать экспериментально, руководствуясь при этом критерием завершения обучения (например, минимизация ошибки или ограничение по времени обучения).

2.2.1.5 Обучение сети

В процессе обучения сеть в определённом порядке просматривает обучающую выборку. Порядок просмотра может быть последовательным, случайным и т. д. Некоторые сети, обучающиеся без учителя (например, сети Хопфилда), просматривают выборку только один раз. Другие (например, сети Кохонена), а также сети, обучающиеся с учителем, просматривают выборку множество раз, при этом один полный проход по выборке называется эпохой обучения. При обучении с учителем набор исходных данных делят на две части – собственно обучающую выборку и тестовые данные; принцип разделения может быть произвольным. Обучающие данные подаются сети для обучения, а проверочные используются для расчета ошибки сети (проверочные данные никогда для обучения сети не применяются). Таким образом, если на проверочных данных ошибка уменьшается, то сеть действительно выполняет обобщение. Если ошибка на обучающих данных продолжает уменьшаться, а ошибка на тестовых данных увеличивается, значит, сеть перестала выполнять

обобщение и просто «запоминает» обучающие данные. Это явление называется переобучением сети или оверфиттингом. В таких случаях обучение обычно прекращают. В процессе обучения могут проявиться другие проблемы, такие как паралич или попадание сети в локальный минимум поверхности ошибок. Невозможно заранее предсказать проявление той или иной проблемы, равно как и дать однозначные рекомендации к их разрешению.

Все вышесказанное относится только к итерационным алгоритмам поиска нейросетевых решений. Для них действительно нельзя ничего гарантировать и нельзя полностью автоматизировать обучение нейронных сетей. Однако, наряду с итерационными алгоритмами обучения, существуют не итерационные алгоритмы, обладающие очень высокой устойчивостью и позволяющие полностью автоматизировать процесс обучения.

2.2.1.6 Проверка адекватности обучения

Даже в случае успешного, на первый взгляд, обучения сеть не всегда обучается именно тому, чего от неё хотел создатель. Известен случай, когда сеть обучалась распознаванию изображений танков по фотографиям, однако позднее выяснилось, что все танки были сфотографированы на одном и том же фоне. В результате сеть «научилась» распознавать этот тип ландшафта, вместо того чтобы «научиться» распознавать танки. Таким образом, сеть «понимает» не то, что от неё требовалось, а то, что проще всего обобщить.

Тестирование качества обучения нейросети необходимо проводить на примерах, которые не участвовали в её обучении. При этом число тестовых примеров должно быть тем больше, чем выше качество обучения. Если ошибки нейронной сети имеют вероятность близкую к одной миллиардной, то и для подтверждения этой вероятности нужен миллиард тестовых примеров. Получается, что тестирование хорошо обученных нейронных сетей становится очень трудной задачей.

2.2.2 Виды нейронных сетей

По характеру обучения различают следующие нейронные сети:

- обучение с учителем – выходное пространство решений нейронной сети известно;

- обучение без учителя – нейронная сеть формирует выходное пространство решений только на основе входных воздействий. Такие сети называют самоорганизующимися;

- обучение с подкреплением – система назначения штрафов и поощрений от среды.

По характеру связей:

- сети прямого распространения (Feedforward). Все связи направлены строго от входных нейронов к выходным. Примерами таких сетей являются перцептрон Розенблатта, многослойный перцептрон, сети Ворда;

- рекуррентные нейронные сети. Сигнал с выходных нейронов или нейронов скрытого слоя частично передается обратно на входы нейронов входного слоя (обратная связь). Рекуррентная сеть Хопфилда «фильтрует» входные данные, возвращаясь к устойчивому состоянию и, таким образом, позволяет решать задачи компрессии данных и построения ассоциативной памяти. Частным случаем рекуррентных сетей являются двунаправленные сети. В таких сетях между слоями существуют связи как в направлении от входного слоя к выходному, так и в обратном. Классическим примером является Нейронная сеть Коско;

- самоорганизующиеся карты. Такие сети представляют собой соревновательную нейронную сеть с обучением без учителя, выполняющую задачу визуализации и кластеризации. Является методом проецирования многомерного пространства в пространство с более низкой размерностью (чаще всего, двумерное), применяется также для решения задач моделирования, прогнозирования и др. Является одной из версий нейронных сетей Кохонена.

Самоорганизующиеся карты Кохонена служат, в первую очередь, для визуализации и первоначального («разведывательного») анализа данных.

2.2.3 Выбор вида нейронной сети

Выбор вида сети будем основывать на следующих критериях:

- заранее известна обучающая выборка (набор исходных кодов на OpenGL с вызовами библиотечных функций). Следовательно, это должна быть сеть, реализующая обучение с учителем;

- решается задача классификации;

- предполагается, что между входными данными имеется ассоциативные связи.

Составим сравнительную характеристику видов нейронных сетей по заданным критериям в виде таблицы 2.

Таблица 2 – Сравнительная характеристика видов нейронных сетей

	Механизм обучения с учителем	Решение задачи классификации	Поиск ассоциативных связей
Сети прямого распространения	+	+	
Рекуррентные нейронные сети	+	+	+
Самоорганизующиеся карты			

Таким образом, наиболее подходящим направлением дальнейших исследований являются рекуррентные нейронные сети.

2.3 Рекуррентные нейронные сети

Рекуррентные нейронные сети – вид нейронных сетей, где связи между элементами образуют направленную последовательность. Благодаря этому появляется возможность обрабатывать серии событий во времени или

последовательные пространственные цепочки. В отличие от многослойных перцептронов, рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины. Поэтому сети RNN применимы в таких задачах, где нечто целостное разбито на части, например: распознавание рукописного текста или распознавание речи. Было предложено много различных архитектурных решений для рекуррентных сетей от простых до сложных.

В последнее время наибольшее распространение получили сеть с долговременной и кратковременной памятью (LSTM) и управляемый рекуррентный блок (GRU).

2.3.1 LSTM

LSTM-сеть – это искусственная нейронная сеть, содержащая LSTM-модули вместо или в дополнение к другим сетевым модулям. LSTM-модуль – это рекуррентный модуль сети, способный запоминать значения как на короткие, так и на длинные промежутки времени. Ключом к данной возможности является то, что LSTM-модуль не использует функцию активации внутри своих рекуррентных компонентов. Таким образом, хранимое значение не размывается во времени, и градиент или штраф не исчезает при использовании метода обратного распространения ошибки во времени при тренировке сети.

LSTM-модули часто группируются в «блоки», содержащие различные LSTM-модули. Подобное устройство характерно для «глубоких» многослойных нейронных сетей и способствует выполнению параллельных вычислений с применением соответствующего оборудования. В формулах ниже каждая переменная, записанная строчным курсивом, обозначает вектор размерности равной числу LSTM-модулей в блоке.

LSTM-блоки содержат три или четыре «вентиля», которые используются для контроля потоков информации на входах и на выходах памяти данных блоков. Эти вентили реализованы в виде логистической функции для

вычисления значения в диапазоне [0; 1]. Умножение на это значение используется для частичного допуска или запрещения потока информации внутрь и наружу памяти. Например, «входной вентиль» контролирует меру вхождения нового значения в память, а «вентиль забывания» контролирует меру сохранения значения в памяти. «Выходной вентиль» контролирует меру того, в какой степени значение, находящееся в памяти, используется при расчёте выходной функции активации для блока (в некоторых реализациях входной вентиль и вентиль забывания воплощаются в виде единого вентиля. Идея заключается в том, что старое значение следует забывать тогда, когда появится новое значение достойное запоминания).

2.3.2 GRU

Управляемые рекуррентные блоки – механизм вентилей для рекуррентных нейронных сетей, представленный в 2014 году. Было установлено, что его эффективность при решении задач моделирования музыкальных и речевых сигналов сопоставима с использованием долгой краткосрочной памяти (LSTM). По сравнению с LSTM у данного механизма меньше параметров, так как отсутствует выходной вентиль.

2.3.3 Выбор вида рекуррентной нейронной сети

Из рассмотренных двух видов рекуррентных нейронных сетей больший интерес для исследования представляет GRU, так как работа LSTM в области поиска пропущенных вызовов функций уже была исследована в статье «Поиск недостающих вызовов библиотечных функций с использованием машинного обучения» [18].

2.4 Механизм нейронной сети

Архитектура нейронной сети представлена на рисунке 9.

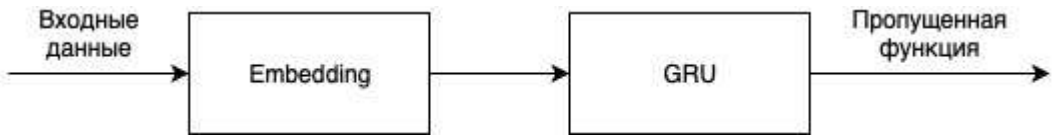


Рисунок 9 – Архитектура нейронной сети

Embedding, или встраивание, преобразует поступающие в него вызовы библиотечных функций в векторное представление. Векторизация позволяет улучшить обучаемость нейронной сети при работе с последовательностями. С помощью векторизации каждый вызов преобразуется в вектор длиной в 32 элемента.

GRU – слой, реализующий рекурсивную нейронную сеть. Данный слой построен на базе модуля управляемого рекуррентного блока (Gated Recurrent Units, GRU).

3 Программная реализация инструмента для поиска пропущенных вызовов функций

3.1 Используемые программные средства

3.1.1 Qt

Qt – кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++ [19]. Есть также «привязки» ко многим другим языкам программирования: Python – PyQt, PySide; Ruby – QtRuby; Java – Qt Jambi; PHP – PHP-Qt и другие.

Со времени своего появления в 1996 году библиотека легла в основу многих программных проектов. Кроме того, Qt является фундаментом популярной рабочей среды KDE, входящей в состав многих дистрибутивов Linux.

Qt позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных систем путём простой компиляции программы для каждой системы без изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования.

Отличительная особенность – использование метаобъектного компилятора – предварительной системы обработки исходного кода. Расширение возможностей обеспечивается системой плагинов, которые возможно размещать непосредственно в панели визуального редактора. Также существует возможность расширения привычной функциональности виджетов, связанной с размещением их на экране, отображением, перерисовкой при изменении размеров окна.

Qt комплектуется визуальной средой разработки графического интерфейса Qt Designer, позволяющей создавать диалоги и формы в режиме WYSIWYG. В поставке Qt есть Qt Linguist – графическая утилита, позволяющая упростить локализацию и перевод программы на многие языки; и Qt Assistant – справочная система Qt, упрощающая работу с документацией по библиотеке, а также позволяющая создавать кроссплатформенную справку для разрабатываемого на основе Qt программного обеспечения. Начиная с версии 4.5.0 в комплект включена среда разработки Qt Creator, которая включает редактор кода, справку, графические средства Qt Designer и возможность отладки приложений. Qt Creator может использовать GCC или Microsoft VC++ в качестве компилятора и GDB в качестве отладчика. Для Windows-версий библиотека комплектуется компилятором, заголовочными и объектными файлами MinGW.

Существуют версии библиотеки для Microsoft Windows, систем класса UNIX с графической подсистемой X11, Android, iOS, Mac OS X, Microsoft Windows CE, QNX, встраиваемых Linux-систем и платформы S60. Идет портирование на Windows Phone и Windows RT. Также идёт портирование на Haiku и Tizen.

Некоторое время библиотека также распространялась ещё в версии Qt/Embedded, предназначенной для применения на встраиваемых и мобильных устройствах, но начиная с середины 2000-х годов она выделена в самостоятельный продукт Qtopia.

Начиная с версии 4.5 Qt распространяется по трём лицензиям:

- Qt Commercial – для разработки программного обеспечения с собственнической лицензией, допускающая модификацию самой Qt без раскрытия изменений;

- GNU GPL – для разработки с открытыми исходниками, распространяемыми на условиях GNU GPL, а также для модификации Qt;

- GNU LGPL – для разработки программного обеспечения с собственнической лицензией.

Исходный код, единый для всех вариантов лицензий, свободно доступен в Git-хранилище, расположенном на Github. Кроме самого исходного кода Qt, там же расположены хранилища сопутствующих библиотек, разрабатываемых авторами библиотеки и сообществом.

До версии 4.0.0 под свободной лицензией распространялись лишь Qt/Mac, Qt/X11, Qt/Embedded, но, начиная с 4.0.0 (выпущенной в конце июня 2005), Qt Software «освободили» и Qt/Windows. При этом существовали сторонние свободные версии Qt/Windows ранее 4.0.0, сделанные на основе Qt/X11.

3.1.2 LLVM

LLVM (Low Level Virtual Machine)– это универсальная система анализа, трансформации и оптимизации программ или, как её называют разработчики, «compiler infrastucture» [20].

В основе LLVM лежит промежуточное представление кода (intermediate representation, IR), над которым можно производить трансформации во время компиляции, компоновки (linking) и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически (JIT-компиляция). LLVM поддерживает генерацию кода для x86, x86-64, ARM, PowerPC, SPARC, MIPS, IA-64, Alpha.

LLVM написана на C++ и портирована на большинство *nix-систем и Windows. Система имеет модульную структуру и может расширяться дополнительными алгоритмами трансформации (compiler passes) и кодогенераторами для новых аппаратных платформ. Пользовательский фронтенд, как правило, линкуется с LLVM и использует C++ API для генерации кода и его преобразований. Однако LLVM включает в себя и standalone утилиты.

Транслятор Си на основе LLVM будет достаточно прост и прямолинеен, но при этом сгенерированный им машинный код по производительности сможет тягаться с последними версиями GCC.

При трансляции высокоуровневых языков – объектно-ориентированных, функциональных, динамических – придётся выполнить гораздо больше промежуточных преобразований, а также написать специализированный рантайм. Но и в этом случае LLVM снимает с разработчика компилятора проблемы кодогенерации для конкретной платформы, берёт на себя большинство независимых от языка оптимизаций – и делает их качественно. Помимо этого, мы получаем готовую инфраструктуру для JIT-компиляции и возможность link-time оптимизации между различными языками, компилируемыми в LLVM.

LLVM пытается достичь баланса между удобством и гибкостью, не навязывая какую-то конкретную парадигму программирования, не ограничивая систему типов.

3.1.3 SQLite

SQLite – компактная встраиваемая СУБД [21]. Слово «встраиваемый» (embedded) означает, что SQLite не использует парадигму клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой библиотеку, с которой программа компонуется, и движок становится составной частью программы. Таким образом, в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа. Простота реализации достигается за счёт того, что перед началом исполнения транзакции записи весь файл, хранящий базу данных, блокируется; ACID-функции достигаются в том числе за счёт создания файла журнала.

Несколько процессов или потоков могут одновременно без каких-либо проблем читать данные из одной базы. Запись в базу можно осуществить только

в том случае, если никаких других запросов в данный момент не обслуживается; в противном случае попытка записи оканчивается неудачей, и в программу возвращается код ошибки. Другим вариантом развития событий является автоматическое повторение попыток записи в течение заданного интервала времени.

В комплекте поставки идёт также функциональная клиентская часть в виде исполняемого файла sqlite3, с помощью которого демонстрируется реализация функций основной библиотеки. Клиентская часть является кроссплатформенной утилитой командной строки.

Благодаря архитектуре движка возможно использовать SQLite как на встраиваемых системах, так и на выделенных машинах с гигабайтными массивами данных.

3.1.4 Keras

Keras – это библиотека для Python с открытым исходным кодом, которая позволяет легко создавать нейронные сети [22]. Библиотека совместима с TensorFlow, Microsoft Cognitive Toolkit, Theano и MXNet. Tensorflow и Theano являются наиболее часто используемыми численными платформами на Python для разработки алгоритмов глубокого обучения, но они довольно сложны в использовании. Оценка популярности фреймворков из статьи «Deep Learning Framework Power Scores 2018» [23] отображена на рисунке 10.

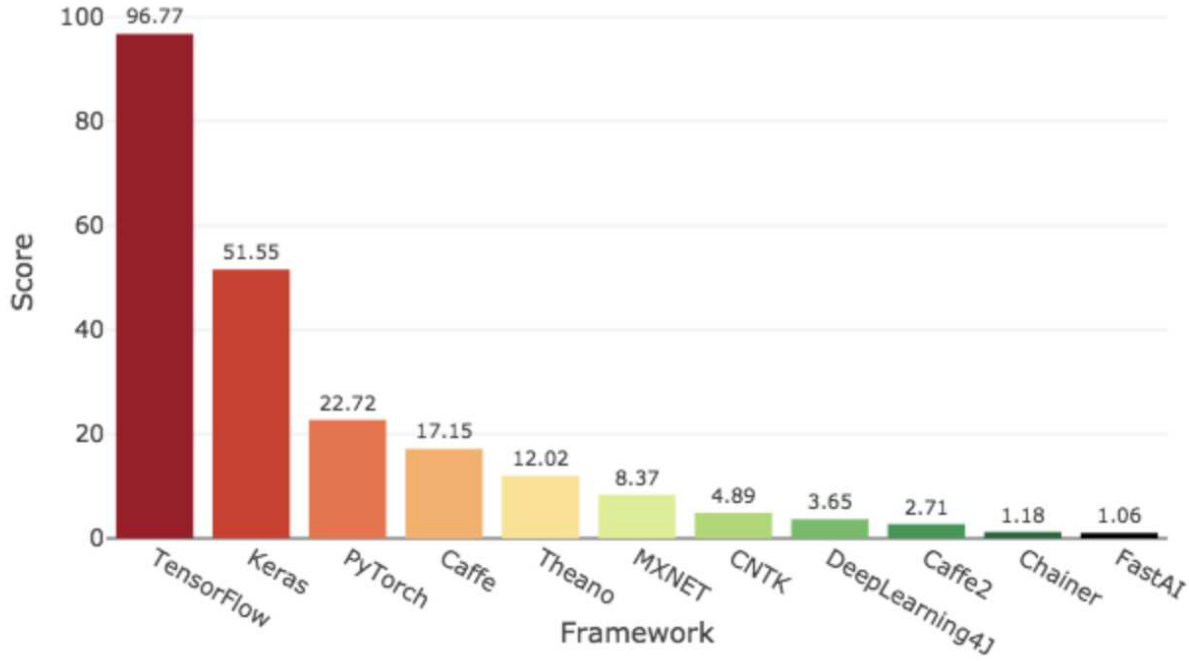


Рисунок 10 – Оценка популярности фреймворков машинного обучения по 7 категориям

Keras, наоборот, предоставляет простой и удобный способ создания моделей глубокого обучения. Ее создатель, François Chollet, разработал ее для того, чтобы максимально ускорить и упростить процесс создания нейронных сетей. Он сосредоточил свое внимание на расширяемости, модульности, минимализме и поддержке Python. Keras можно использовать с GPU и CPU; она поддерживает как Python 2, так и Python 3. Keras компании Google внесла большой вклад в коммерциализацию глубокого обучения и искусственного интеллекта, поскольку она содержит современные алгоритмы глубокого обучения, которые ранее были не только недоступными, но и непригодными для использования.

Главные преимущества Keras:

- дружественность к пользователю. Keras имеет простой, согласованный интерфейс, оптимизированный для случаев общего применения. Он обеспечивает четкую и эффективную обратную связь для пользовательских ошибок;

- модулярный и компонуемый. Keras модели изготовлены посредством связывания конфигурируемых строительных блоков вместе, с несколькими ограничениями;

- простой для расширения. Можно писать пользовательские строительные блоки для выражения новых идей для исследования. Можно создавать новые слои, функции потери, и разрабатывать современные модели.

3.2 Разработка нейронной сети с использованием Keras

3.2.1 Сбор данных для обучения

Обучение любой модели в машинном обучении начинается с данных. В ходе сбора данных необходимо выполнить следующие задачи:

- получить список файлов с исходным кодом;

- извлечь из них все вызовы функций;

- определить список видов вызываемых библиотечных функций;

- подготовить наборы данных в виде пары (список вызовов, пропущенная функция).

В качестве источника данных будем использовать ресурс с обучающими примерами OpenGL [24]. Алгоритм загрузки файлов с исходным кодом продемонстрирован в листинге 3.

Листинг 3 – Алгоритм загрузки файлов

```
1. Алгоритм: получение списка файлов с исходным
2. Вход: url источник с файлами
3. Выход: файлы с исходным кодом из указанного источника
4. htmlCode = получитьКодСтраницы(url)
5. htmlTree = преобразоватьКодВДерево(htmlCode)
6. links = получитьСписокССылокНаФайлы(htmlTree)
7. для каждого k в диапазоне от 0 до длина(links) – 1:
8.       fileLink = извлечьСвойствоHref(links[k])
9.       fileName = извлечьНазваниеФайла(fileLink)
10.      сохранитьФайл(fileLink, fileName)
```

По результатам работы алгоритма из источника с исходным [19] было получено 155 примеров.

Для извлечения набора вызовов преобразуем каждый полученный файл в промежуточное представление LLVM с помощью команды *clang-3.5 -c file.c -O0 -emit-llvm -S -o output.ll*, где:

- *file.c* – путь до файла с исходным кодом;

- *O0* – уровень оптимизации (без оптимизации);

- *emit-llvm -S* – указание clang-у сгенерировать файл llvm в текстовом виде (на ассемблере LLVM);

- *output.ll* – файл с промежуточным представлением.

В преобразованном файле нам необходимо найти пользовательскую функцию и получить из нее все вызовы других функций. Общий вид объявленной пользователем функции в LLVM представлен в листинге 4.

Листинг 4 – Общий вид объявленной функции в LLVM

```
1. define returntype @funcName(args) #attrs {
2.         …
3. }
```

Структура функции:

- returntype – возвращаемое функцией значение;

- funcName – название функции;

- args – аргументы функции;

- #attrs – номер перечисления списка параметров функции в LLVM.

Составим регулярное выражение для определения функции. Очевидно, что строка должна начинаться с ключевого слова define, содержать символы @, (,), #, }. Строка также должна удовлетворять следующим условиям:

- после ключевого слова *define* должен идти возвращаемый тип, который может содержать латинские буквы, цифры и знак *%*;

- после символа @ должно идти название функции;

- между символами (и) могу содержаться аргументы, перечисленные через запятую;

- после знака # должен идти номер списка параметров функции.

В итоге получаем следующее регулярное выражение, представленное в листинге 5.

Листинг 5 – Регулярное выражение для определения начала пользовательской процедуры

```
1. R"^define [%a-zA-Z0-9]* @(\w)* \((\w)*(,{0,1}))*\) #[0-9]* {$"
```

После нахождения пользовательской процедуры необходимо извлечь все вызовы функций из нее. Пример вызова функции представлен в листинге 6.

Листинг 6 – Пример вызова функции в LLVM

```
1. %retval = call i32 @test(i32 %argc)
```

Вызов функции обязательно содержит ключевое слово *call* и идентификатор функции. Регулярное выражение для определения, является ли строка вызовом функции представлено в листинге 7.

Листинг 7 – Регулярное выражение для определения вызова функции

```
1. R"(.)+(call)([^@])*(@)(.)*"
```

Алгоритм для получения списка видов вызываемых функций представлен в листинге 8.

Листинг 8 – Алгоритм получения набора функций

```
1. Алгоритм: получение набора вызовов функций из файла
2. Вход: llvm представление исходного файла в виде списка строк lines
3. Выход: список видов библиотечных функций
4. funcSet = []
5. для каждого k в интервале от 0 до длина(lines) – 1:
6.       если lines[k] соответствует шаблону вызова то:
```

```
7.          funcName = извлечьНазваниеФункции(line)
8.          если funcName не присутствует в funcSet то:
9.              добавить(funcName, funcSet)
10. вернуть funcSet
```

После извлечения списка функций необходимо также исключить все функции, не входящие в состав OpenGL. После работы алгоритма в листинге 6 и удаления всех не-OpenGL функций был получен список из 49 элементов. Отсюда получаем, что при решении задачи классификации мы будем иметь 50 классов (по 1-му классу на полученную функцию и один класс для случая, когда пропущенная функция отсутствует).

Далее нам необходимо сформировать выборку для обучения. Каждый элемент выборки должен иметь вид кортежа, состоящего из:

- списка вызовов функций;

- идентификатора класса, соответствующего пропущенной функции/отсутствию пропущенной функции;

Формирование выборки должно состоять из двух этапов:

- извлечение списка вызовов библиотечных функций из файла;

- генерация набора примеров для обучения из полученного списка вызовов.

Алгоритм для получения списка вызовов функций из файла представлен в листинге 9.

Листинг 9 – Алгоритм получения списка функций

```
1. Алгоритм: получение списка вызовов функций из файла
2. Вход: llvm представление исходного файла в виде списка строк lines
3. Выход: список вызов библиотечных функций
4. callList = []
5. для каждого k в интервале от 0 до длина(lines) – 1:
6.      если lines[k] соответствует шаблону вызова то:
7.          funcName = извлечьНазваниеФункции(line)
8.          если funcName – вызов библиотечной функции то:
9.              добавить(funcName, callList)
10. вернуть callList
```

Алгоритм генерации примеров представлен в листинге 10.

Листинг 10 – Алгоритм генерации примеров для обучения

```
1. Алгоритм: Алгоритм генерации примеров для обучения
2. Вход: список вызовов callList
3. Выход: список примеров для обучения exampleSet, размер – длина(callList) + 1
4. exampleSet = []
5. для каждого k в интервале от 0 до длина(callList) – 1:
6.     callListCopy = клонировать(callList)
7.     deletedCall = удалитьЭлемент(k, callListCopy)
8.     exampleTuple = (callListCopy, deletedCall)
9.     добавить(exampleTuple, exampleSet)
9. call = ‘ ‘ // означает, что пропущенных вызовов нет
10. exampleSet = (callList, call)
11. вернуть exampleSet
```

По результатам работы алгоритма из 155 файлов исходного кода было получено 3177 примеров для обучения.

3.2.2 Разработка и обучение модели нейронной сети

Процесс работы с моделью в keras состоит из следующих этапов:

- формирование модели из слоев;

- компиляция модели;

- обучение модели.

В соответствии с разработанной архитектурой нейронной сети в пункте 2.4, построим модель сети в keras. Механизм построения состоит из следующих этапов:

- инициализация объекта класса *Sequential*;

- наполнение созданного объекта слоями с помощью встроенного метода *add*.

Каждый слой представляет из себя объект одного из предопределенных классов. В keras реализовано более 100 видов слоев. Из них нам понадобятся 2 вида – Embedding и GRU. Синтаксис инициализации слоя Embedding представлен в листинге 11.

Листинг 11 – Синтаксис инициализации слоя Embedding

```
keras.layers.Embedding(
    input_dim,
    output_dim,
    embeddings_initializer='uniform',
    embeddings_regularizer=None,
    activity_regularizer=None,
    embeddings_constraint=None,
    mask_zero=False,
    input_length=None
)
```

Аргументы:

- input_dim: int > 0 – количество входов слоя;

- output_dim: int >= 0 – количество выходов слоя;

- embeddings_initializer – функция первоначальной инициализации весов слоя;

- embeddings_regularizer – функция, применяемая в случае использования механизма штрафов для нейронов входного слоя;

- activity_regularizer – функция, применяемая в случае использования механизма штрафов для нейронов выходного слоя;

- embeddings_constraint – функция, описывающая ограничения на входной слой;

- mask_zero – флаг, определяющий необходимость замены входных нулей на специальные «пропуски». Это полезно при использовании рекуррентных слоев, которые могут принимать входные данные переменной длины;

- input_length – размерность входных данных.

Синтаксис инициализации слоя GRU представлен в листинге 12.

Листинг 12 – Синтаксис инициализации слоя GRU

```
keras.layers.GRU (
    units,
    activation='tanh',
    recurrent_activation='hard_sigmoid',
    use_bias=True,
    kernel_initializer='glorot_uniform',
    recurrent_initializer='orthogonal',
    bias_initializer='zeros',
    dropout=0.0,
    recurrent_dropout=0.0,
    return_sequences=False,
    go_backwards=False,
)
```

Аргументы:

- units – размерность данных на выходе;

- activation – функция активации, по умолчанию гиперболический тангенс;

- recurrent_activation – функция активации на каждом шаге рекуррентного подбора параметров модели;

- use_bias – флаг, указывающий на необходимость использования смещения в функции активации;

- kernel_initializer – функция первоначальной инициализации весов слоя;

- recurrent_initializer – функция инициализации весов слоя на каждом шаге рекуррентного подбора параметров модели;

- bias_initializer – функция инициализации смещения значений функции активации;

- dropout – доля не учитываемых связей модели при получении выхода модели (функция «забывания» информации);

- recurrent_dropout – доля не учитываемых связей модели при получении выхода модели на каждом шаге рекуррентного подбора параметров модели;

- return_sequences – определяет формат выхода слоя: лучшее значение или набор всех значений;

- go_backwards – определяет необходимость инвертирования данных на входе.

В Keras основная работа по настройке модели заключается в подборе архитектуры сети (набора слоев) и параметров каждого из слоев.

Код по настройке модели с оптимальными параметрами приведен в листинге 13.

Листинг 13 – Настройка модели нейронной сети

```
1. model = Sequential()
2. model.add(Embedding(50, 32, input_length=36))
3. model.add(GRU(50, activation=softmax, recurrent_activation='hard_sigmoid',
4.                 dropout=0.1, recurrent_dropout=0.1))
```

После настройки модели ее необходимо скомпилировать. Для этого используется метод *compile* объекта класса Sequential. Код компиляции модели представлен в листинге 14.

Листинг 14 – Компиляция построенной модели

```
1. model.compile(
2.       optimizer='rmsprop',
3.       loss='categorical_crossentropy',
4.       metrics=['accuracy']
5. )
```

Перед началом использования, модель надо обучить. Производится это с помощью метода *fit* у модели. Код обучения модели представлен в листинге 15:

Листинг 15 – Обучение модели

```
1. model.fit(
2.       train_data,
3.       train_label,
4.       epochs=100,
5.       batch_size=16,
6.       validation_data=(test_data, test_label)
7. )
```

Описание передаваемых параметров в функцию:

- *train_data* – набор списков вызовов функций (примеры);

- *train_label* – набор классов, представляющих пропущенный вызов/его отсутствует в вышеуказанных списках вызовов;

- *epochs* – количество эпох при проведении процесса обучения. Эпоха – это процесс, по результатам которого в алгоритм подбора параметров модели будет полностью передана обучающая выборка;

- *batch_size* – размер пакетов, на которые будет разделена выборка при обучении модели в рамках эпохи;

- *validation_data* – тестовая выборка. Представляет из себя кортеж из набора списков вызовов функций и соответствующих им классов.

Перед обучением набор примеров, полученных после работы алгоритма из листинга 10, был разделен на обучающую и тестовую выборку в соотношении 9:1 (тестовая выборка составила 10% от всех примеров).

Метод *fit* возвращает объект истории обучения, в котором содержится информация по эпохам о точности модели на обучающей и тестовой выборках. Визуализация точности модели на тестовой выборке после подбора оптимальных параметров модели Sequential представлена на рисунке 11.

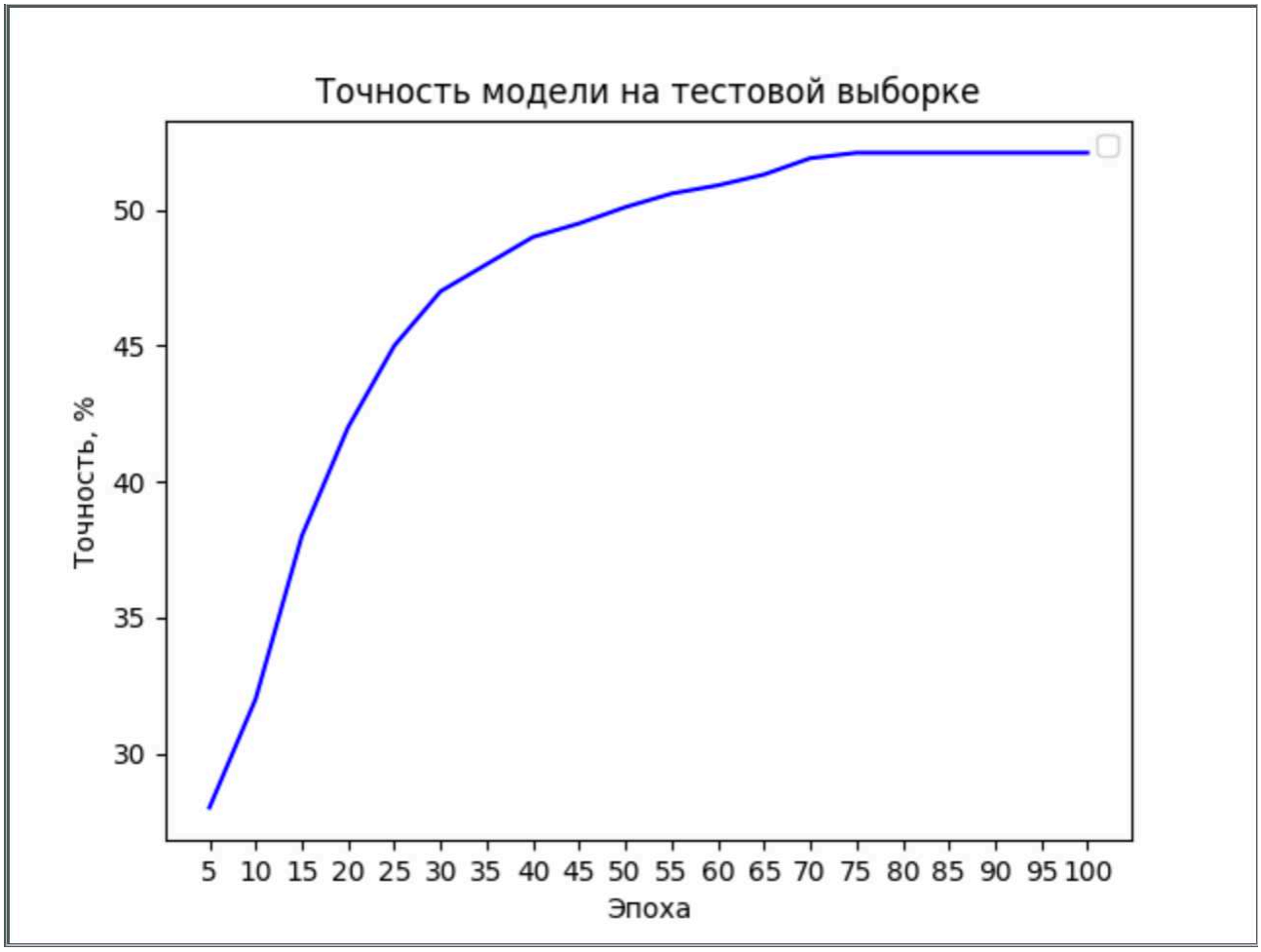


Рисунок 11 – Точность модели на тестовой выборке по эпохам

Из графика видно, что, начиная с 80 эпохи, точность модели не увеличивалась и достигла приблизительно 50 %. В конце обучения получаем точность модели на тестовой выборке 52,1 %. Данный результат не является приемлемым для конечного пользователя. Для выяснения причин столько низкого результата проанализируем входные данные модели. Ниже приведены предполагаемые причины неудачного обучения модели:

- количество вызовов функций в примерах – от 10 до 36. Для обучения каждый набор был дополнен до 36 элементов нулями. Вследствие этого в обучающих данных образуется «пустота» из нулей, которая может сказываться на обучении. Данную проблему решить не удалось, так как произвести выравнивание наборов по количеству вызовов в них невозможно в автоматическом режиме. Также, при выравнивании наборов могут потеряться ассоциативные связи между вызовами, что может привести только к ухудшению

результатов обучения, так как в основе модели заложены рекуррентные нейронные сети;

- распределение функций не является равномерным. При подсчете количества вызовов каждой из встречающихся функций, выяснилось, что самая часто встречающаяся функция (*glEnable*) вызывалась 2648 раз, а самая редко встречающаяся функия (*loadDDS*) – всего 15 раз. При обучении важно, чтобы классы были распределены равномерно. Однако в реальных условия этого добиться невозможно.

Рассмотрим работу модели при решении задачи бинарной классификации – определение факта наличия пропущенного вызова. Это можно определить по количеству ложноположительных (функция пропущена не была, но модель обнаружила пропуск) и ложноотрицательных (функция была пропущена, но модель этого не обнаружила) срабатываний модели. График доли ложноположительных и ложноотрицательных срабатываний модели изображен на рисунке 12.

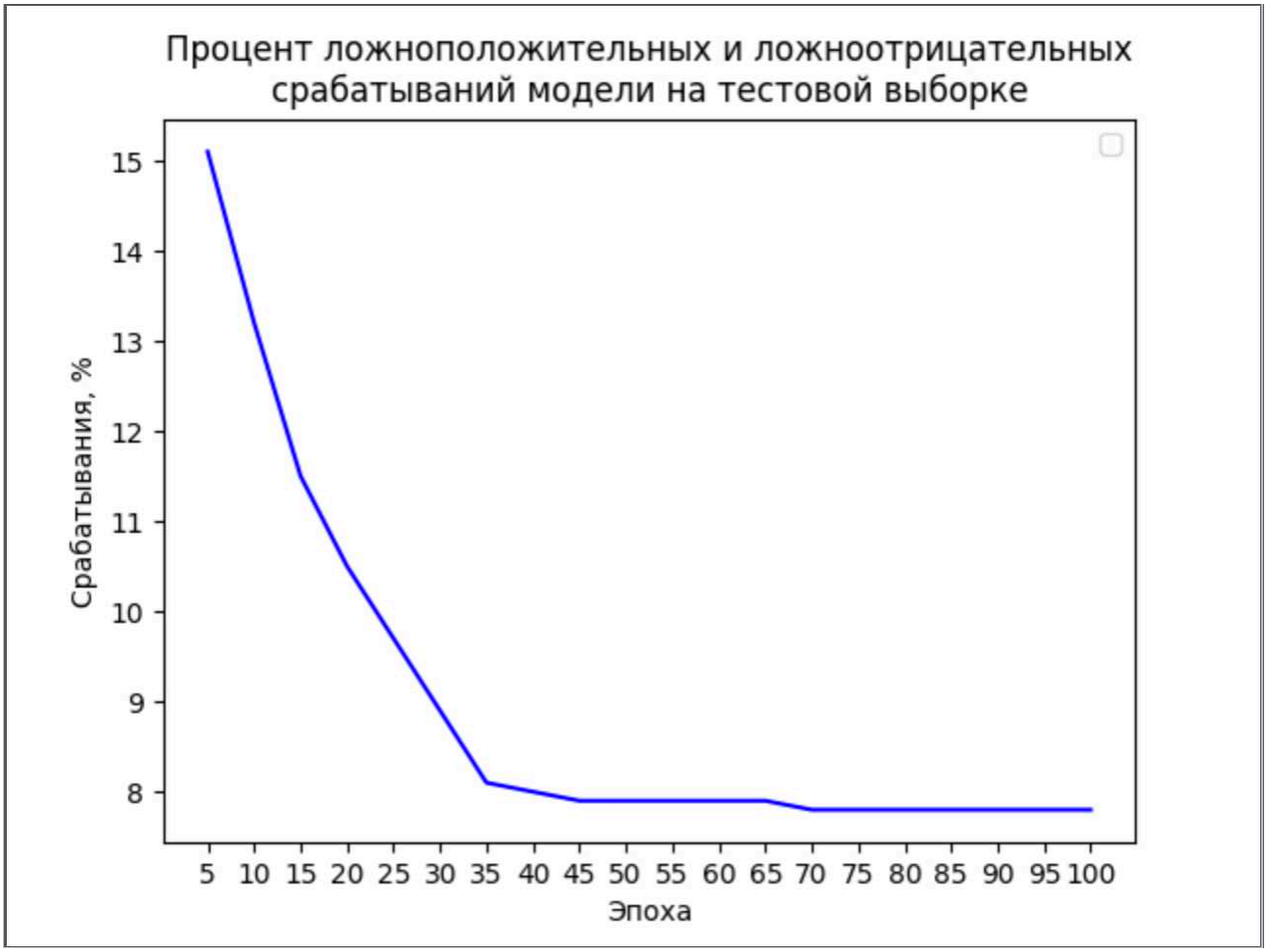


Рисунок 12 – График доли ложноположительных и ложноотрицательных срабатываний по эпохам

Из полученных результатов получаем, что точность модели при решении задачи бинарной классификации – 92,2 %. Данный результат является приемлемым, однако такой подход не решает исходной задачи.

Для повышения точности работы модели можно воспользоваться следующим способом – в качестве результата модели мы будем брать не самый вероятный результат, а первые *k* результатов. Для определения оптимального k произведем оценку top-k полученных результатов. Результаты оценки приведены на рисунке 13.

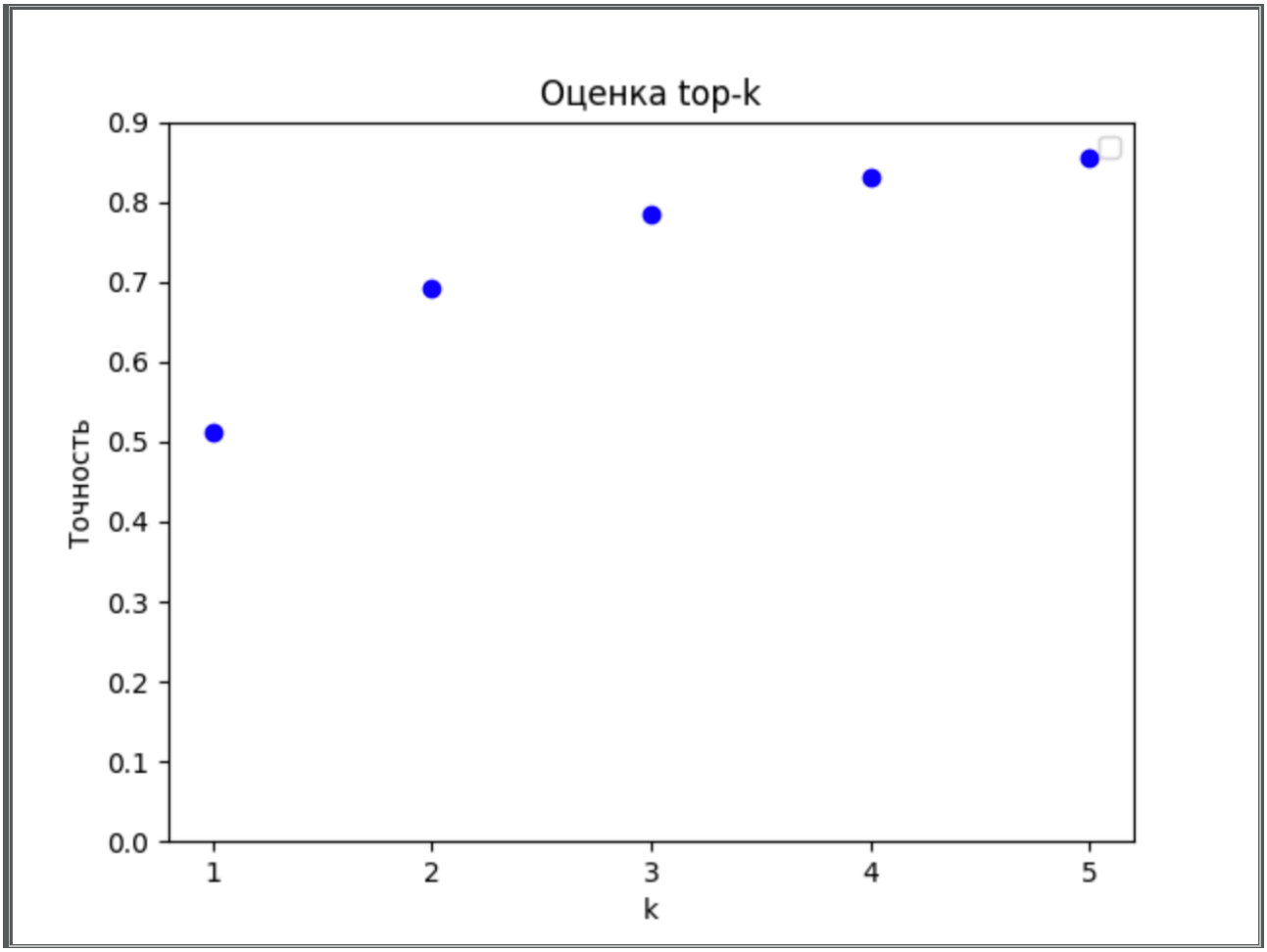


Рисунок 13 – Оценка top-k

Из оценки получаем, что, взяв первые 4 варианта, предлагаемых моделью, мы получаем точность результата в 83,2 %, что является приемлемым результатом. Поэтому, при разработке визуального интерфейса должно быть учтено требования показа пользователю первых 4 результатов, предлагаемых моделью.

3.3 Разработка графического-пользовательского интерфейса

Разрабатываемый пользовательский интерфейс должен предоставлять пользователю возможность выполнять следующие функции:

- загрузка файла на проверку;
- просмотр отчета о проверке;
- просмотр истории отчетов о проверке.

На рисунке 14 изображено главное окно инструмента.

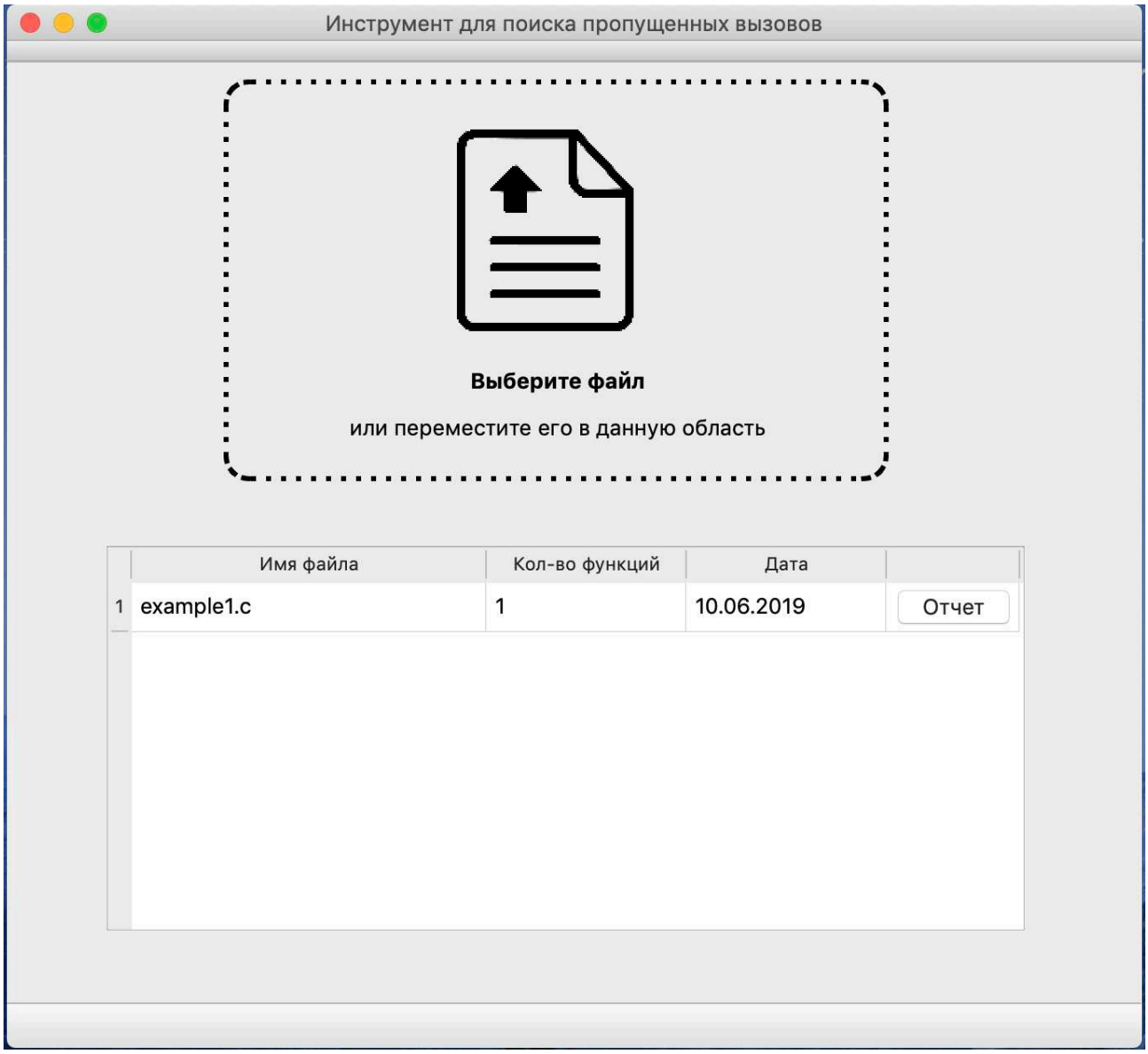


Рисунок 14 – Главное окно инструмента

Главное окно инструмента состоит из двух частей: область для загрузки файла (рисунок 15) и таблица с историей проверок (рисунок 17).

Рисунок 15 – Область для загрузки файла

Загрузить файл пользователь может следующими способами:

- перетащить иконку файла в область для загрузки;

- кликнуть один раз левой кнопкой мыши по области загрузки с последующим выбором файла в стандартном окне выбора файла в операционной системе.

После выбора пользователем файла программ приступает к поиску пропущенных вызовов. Результат поиска отображается в новом окне (рисунок 16).

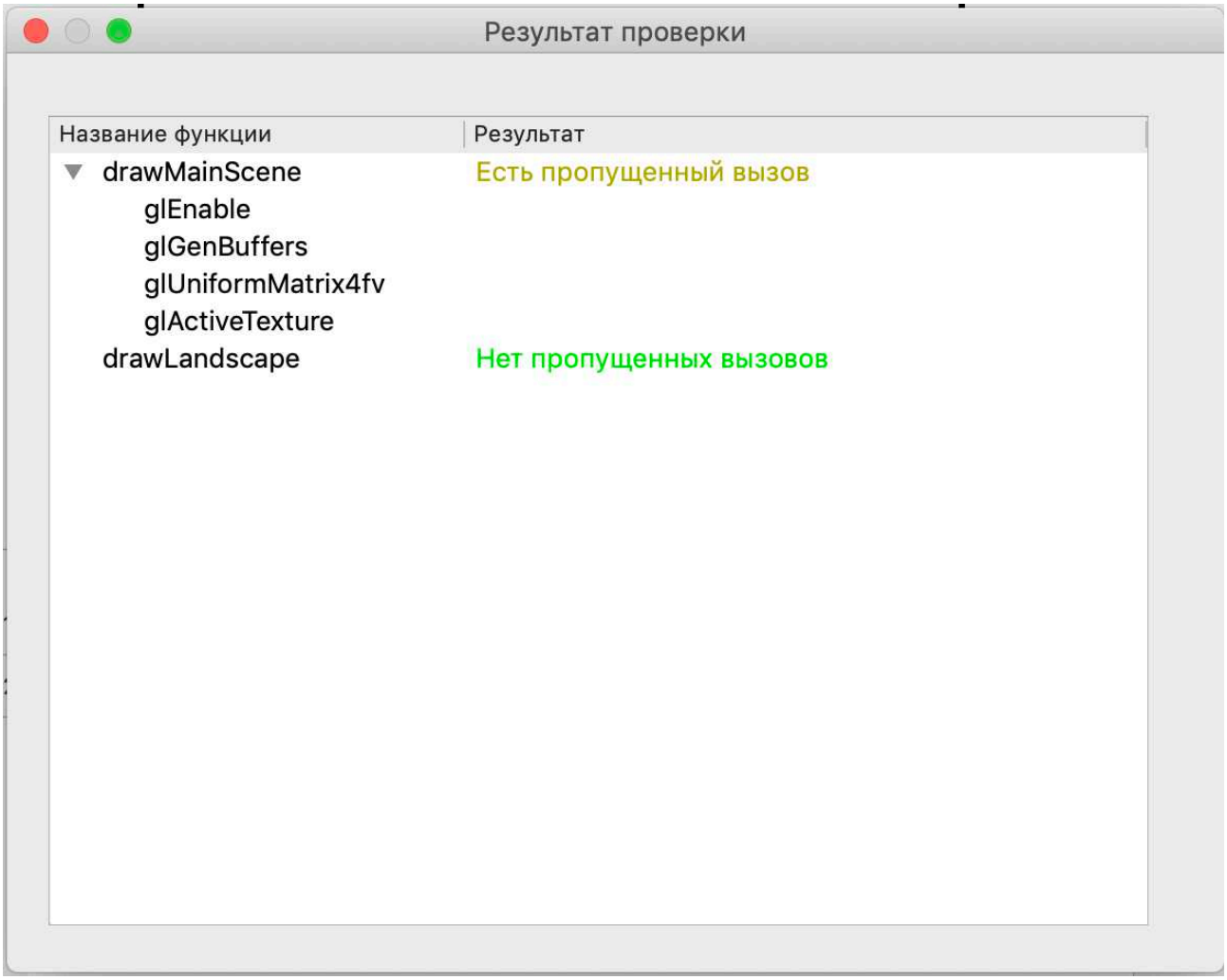


Рисунок 16 – Результаты поиска пропущенных функций

При поиске программа собирает список функций, объявленных пользователем, и для каждой функции определяет наличие/отсутствие пропущенного вызова. В случае наличия пропущенного вызова программа предлагает набор из четырех возможных вариантов функций.

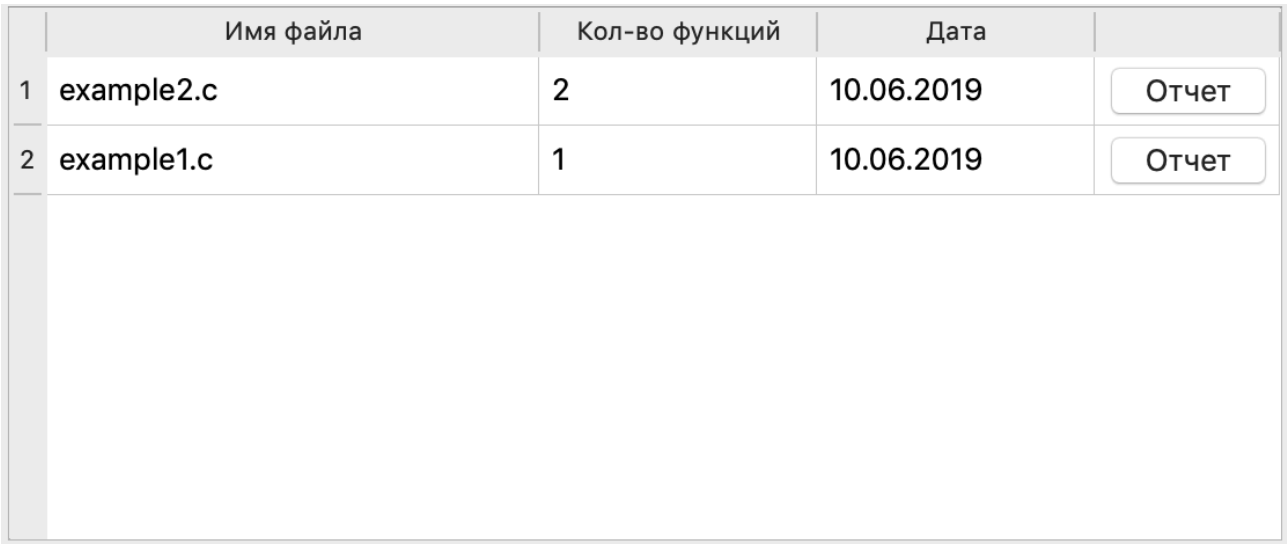


Рисунок 17 – История проверок

В таблице истории проверок отображается следующая информация:

- название проверенного файла;
- количество обнаруженных пользовательских функций;
- дата проверки.

Также, при нажатии на кнопку «Отчет» открывается окно с результатами ранее осуществленной проверки файла. Интерфейс аналогичен изображенному на рисунке 16.

3.4 Апробация инструмента для поиска пропущенных вызовов функций

Перед использованием инструмента необходимо убедиться, что:

- все заявленные функции работают корректно;
- все ошибки обрабатываются корректно.

Для проверки инструмента были определены тест-кейсы, описанные в таблице 3.

Таблица 3 – Описание тест-кейсов

№	Действия	Ожидаемый результат	Результат теста
1	Перенести иконку файла с исходным кодом в область для загрузки.	Окно с результатами поиска.	Тест пройден
2	Нажать на область для загрузки левой кнопкой мыши.	Окно выбора файла.	Тест пройден
3	- Нажать на область для загрузки левой кнопкой мыши; - в открывшемся окне выбрать файл.	Окно с результатами поиска по выбранному файлу.	Тест пройден
4	Перенести иконку файла, не содержащего код на языке c/c++, в область для загрузки.	Сообщение об ошибке «Загруженный файл не содержит код на языке c/c++, либо код некорректен».	Тест пройден
5	Перенести иконку файла, содержащего некорректный код на языке c/c++, в область для загрузки.	Сообщение об ошибке «Загруженный файл не содержит код на языке c/c++, либо код некорректен».	Тест пройден
6	- Убедиться, что на рабочей машине отсутствует интерпретатор python; - запустить программу.	Сообщение об ошибке «Не обнаружен интерпретатор python».	Тест пройден
7	- Загрузить файл с пропущенным вызовом функции; - перезапустить приложение.	- В отчете отображается информация, что в файле пропущен вызов функции; - после процедуры поиска в таблице истории появляется запись, содержащая название загруженного файла, значение 1 в столбце «кол-во функций», текущую дату; - после перезагрузки приложения добавленная строка не исчезает;	Тест пройден

Окончание таблицы 3

№	Действия	Ожидаемый результат	Результат теста
		- при нажатии кнопки «отчет» в появившейся строке отображается окно с теми же результатами, что и сразу же после проверки.	
8	Загрузить файл без пропущенного вызова.	В отчете отображается информация, что в файле нет пропущенных вызовов функций.	Тест пройден
9	Нажать кнопку «отчет» в строке таблицы, появившейся по результатам проверке тест кейса №3.	Открывается окно с результатами проверки, идентичными результатам после процедуры поиска.	Тест пройден

По результатам тестирования все тест-кейсы были успешно пройдены.

ЗАКЛЮЧЕНИЕ

В ходе данной работы был произведен обзор существующих решений по поиску пропущенных вызовов функций. По результатам обзора было определено, что на данный момент в открытых источниках не существует инструментальных средств для поиска пропущенных вызовов функций.

Предложено в качестве метода машинного обучения для решения задачи поиска пропущенных вызовов использование рекуррентных нейронных сетей на основе управляемого рекуррентного блока.

На основе примеров исходного кода были сформированы обучающая и тестовая выборка. Была разработана и обучена модель искусственной нейронной сети и подобраны оптимальные параметры модели. На основе построенной модели был разработано и протестировано инструментальное средство для поиска пропущенных вызовов.

На основании полученных результатов была опровергнута гипотеза о возможности решения задачи поиска пропущенных вызовов как задачи многоклассовой классификации из-за низкой точности получаемых результатов – 52,1 %. Однако, при рассмотрении не одного, а первых четырех результатов модели точность является приемлемой – 83,2 %.

Возможные направления улучшения разработанного инструментального средства:

- функция дообучения модели на основании корректировок от пользователя;

- указание директории для анализа присутствующих в ней файлов в режиме online;

- организация единого сервера с развернутой моделью искусственной нейронной сети для централизации процесса дообучения;

- обнаружения нескольких пропущенных вызовов функций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Карпович, Е. Е. Методология разработки программного обеспечения: электронное пособие / Е. Е. Карпович. – Москва : Центр дистанционного обучения. НИТУ «МИСиС», 2015. – 80 с.

2 Дроботун, Е. Б. Надежность программного обеспечения. виды и критичность ошибок / Е. Б. Дроботун // Информационно-вычислительные технологии в науке / Военная академия воздушно – космической обороны. – Тверь. – 2009. – С. 55–59.

3 Ковалев, И. В. Анализ проблем в области исследования надежности программного обеспечения: многоэтапность и архитектурный аспект / И. В. Ковалев // Вестник СибГАУ. – 2014. – № 3. – С. 78–92.

4 Липаев, В. В. Программная инженерия. Методологические основы / В. В. Липаев. – Москва : ТЕИС, 2006. – 605 с.

5 ГОСТ 27.002-2015. Надежность в технике. Основные понятия. Термины и определения. – Введ. 01.03.2015. – Москва : Стандартинформ, 2015. – 23 с.

6 ГОСТ 19.004-80. ЕСПД. Термины и определения. – Введ. 01.07.1981. – Москва : Стандартинформ, 1981. – 3 с.

7 Королев, А. Подсчет себестоимости часа разработки программного обеспечения [Электронный ресурс] // Хабр. - Москва, 2015. - Режим доступа: https://habr.com/ru/post/256537.

8 Якимов, И. А. Оптимизация читаемости порождаемых при символьных вычислениях тестов / И. А. Якимов, А. С. Кузнецов, А. М. Скрипачев // Сибирский журнал науки и технологий. – 2019. – Т. 20, № 1. – С. 35–41.

9 Макконнелл, С. Совершенный код. Практическое руководство по разработке программного обеспечения / С. Макконнелл. Пер. с англ. – Москва : Издательство «Русская редакция», 2010. – 896 с.

10 Martin Monperrus. Detecting Missing Method Calls in Object-Oriented Software / Martin Monperrus, Marcel Bruch, Mira Mezini // Proceedings of the 24th European Conference on Object-Oriented Programming. – 2010. – P. 25–28.

11 Zhenmin Li. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code / Zhenmin Li, Yuanyuan Zhou // European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. – 2005. – P. 10.

12 Флах, П. Машинное обучение. Наука и искусство построения алгоритмов, которые извлекают знания из данных / П. Флах. Пер. с англ. – Москва : Издательство «ДМК-Пресс», 2015. – 400 с.

13 Gösta Grahne. Efficiently Using Prefix-trees in Mining Frequent Itemsets / Gösta Grahne, Jianfei Zhu // International Conference on Database and Expert Systems Applications. – 2012. – P. 453–476.

14 Айвазян, С. А. Прикладная статистика: классификация и снижение размерности / С. А. Айвазян, В. М. Бухштабер, И. С. Енюков, Л. Д. Мешалкин. – Москва : Финансы и статистика. 1989. – 607 с.

15 Бринк Х. Машинное обучение / Х. Бринк, М. Феверолф, Д. Ричардс. Пер. с англ. – Санкт-Петербург : Издательство «Питер», 2017. – 336 с.

16 ГОСТ 19.701-90. ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения. – Введ. 01.01.1992. – Москва : Стандартинформ, 1992. – 22 с.

17 Николенко, С. И. Глубокое обучение. Погружение в мир нейронных сетей / С. И. Николенко, Е. В. Архангельская, А. А. Кадурин. – Санкт-Петербург : Издательство «Питер», 2018. – 480 с.

18 Якимов, И. А., Поиск недостающих вызовов библиотечных функций с использованием машинного обучения / И. А. Якимов, А. С. Кузнецов // Труды ИСП РАН. – 2017. – Т. 29, № 6. – С. 117–134.

19 Qt | Cross-platform software development for embedded & desktop / The Qt company // Qt Group [сайт]. – Helsinki, 2019. – Режим доступа: https://www.qt.io.

20 The LLVM Compiler Infrastructure [Электронный ресурс] : сайт содержит полную информацию о продукте LLVM. – Режим доступа: https://llvm.org.

21 SQLite Home Page [Электронный ресурс] : сайт содержит полную информацию о системе управления базами данных SQLite. – Режим доступа: https://www.sqlite.org/index.html.

22 Keras: The Python Deep Learning library : сайт содержит информацию об открытой нейросетевой библиотеке Keras. – Режим доступа: https://keras.io.

23 Hale, J. Deep Learning Framework Power Scores 2018 [Электронный ресурс] / J. Hale // Medium. – San Francisco, 2018. – Режим доступа: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a.

24 OpenGL – GLUT Program Sample Code [Электронный ресурс] : архив содержит примеры исходного кода с использованием библиотеки OpenGL. – Режим доступа: https://www.opengl.org/archives/resources/code/samples/glut_examples/examples/examples.html.

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
Кафедра «Информатика»

УТВЕРЖДАЮ
Заведующий кафедрой
_____ А. С. Кузнецов
«09» 07 2019 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Инструментальное средство для выявления пропущенных вызовов библиотеки OpenGL с использованием методов машинного обучения

09.04.04 Программная инженерия

09.04.04.01 Программное обеспечение вычислительной техники и автоматизированных систем

Научный руководитель _____ 08.07.19 доцент, к.т.н. А. С. Кузнецов

Выпускник _____ 08.07.19 А. М. Скрипачев

Рецензент _____ 9.07.19 доцент, к.т.н. Е. П. Моргунов

Красноярск 2019